JBoss Enterprise Application Platform 5.0
JBoss Cache Benutzerhandbuch

---

zum Gebrauch mit der JBoss Enterprise Application Platform 5.0
Ausgabe 2.0

Manik Surtani
Brian Stansberry
Galder Zamarreño
Mircea Markus

JBoss Enterprise Application Platform 5.0 JBoss Cache Benutzerhandbuch

---

zum Gebrauch mit der JBoss Enterprise Application Platform 5.0
Ausgabe 2.0

Manik Surtani
manik@jboss.org

Brian Stansberry
brian.stansberry@jboss.com

Galder Zamarreño
galder.zamarreno@jboss.com

Mircea Markus
mircea.markus@jboss.com

Rechtlicher Hinweis

Copyright © 2009 Red Hat, Inc.

This document is licensed by Red Hat under the Creative Commons Attribution-ShareAlike 3.0 Unported License. If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, MetaMatrix, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack Logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Zusammenfassung

Bei diesem Buch handelt es sich um das Benutzerhandbuch für den Cache der JBoss Enterprise Application Platform.

Inhaltsverzeichnis

Vorwort **6**

Teil I. Einführung in den JBoss Cache **7**

Kapitel 1. Überblick **8**
1.1. Was ist JBoss Cache? 8
1.1.1. Und was ist ein POJO Cache? 8
1.2. Kurzfassung der Features 8
1.2.1. Cachen von Objekten 8
1.2.2. Lokale und geclusterte Modi 9
1.2.3. Geclusterte Caches und Transaktionen 9
1.2.4. Thread-Sicherheit 9
1.3. Anforderungen 10
1.4. Lizenz 10

Kapitel 2. Benutzer-API **11**
2.1. API-Klassen 11
2.2. Instanziierung und Start des Caches 11
2.3. Cachen und Abrufen von Daten 12
2.3.1. Organisieren Ihrer Daten und verwenden der Knotenstruktur 13
2.4. Die Fqn-Klasse 14
2.5. Stoppen und Zerstören des Caches 14
2.6. Cache-Modi 15
2.7. Hinzufügen eines Cache Listeners – Registrieren für Cache-Ereignisse 15
2.7.1. Synchrone und asynchrone Benachrichtigungen 17
2.8. Verwenden von Cache Loadern 17
2.9. Verwenden von Eviction-Richtlinien 18

Kapitel 3. Konfiguration **20**
3.1. Konfigurationsüberblick 20
3.2. Erzeugen einer Konfiguration 20
3.2.1. Analyse einer XML-basierten Konfigurationsdatei 20
3.2.2. Überprüfen von Konfigurationsdateien 20
3.2.3. Befehlsorientierte Konfiguration 21
3.2.4. Verwenden eines IOC-Frameworks 21
3.3. Zusammensetzung eines Configuration-Objekts 21
3.4. Dynamische Rekonfiguration 22
3.4.1. Überschreiben der Konfiguration mittels der Option-API 23

Kapitel 4. Batching-API **24**
4.1. Einführung 24
4.2. Batching-Konfiguration 24
4.3. Batching-API 24

Kapitel 5. Implementierung von JBoss Cache **25**
5.1. Eigenständiger Einsatz / Befehlsorientierte Implementierung 25
5.2. Via JBoss Microcontainer (JBoss AS 5.x) 25
5.3. Automatisches Binden an JNDI in JBoss AS 27
5.4. Informationen zur Verwaltung während der Laufzeit 27
5.4.1. JBoss Cache MBeans 27
5.4.2. CacheJmxWrapper beim MBeanServer registrieren 27
5.4.2.1. Befehlsorientierte Registrierung mit einer Cache-Instanz 28
5.4.2.2. Befehlsorientierte Registrierung mit einer Konfigurationsinstanz 28

5.4.2.3. JMX-basiertes Deployment in JBoss AS (JBoss AS 5.x) 28
5.4.3. JBoss-Cache-Statistiken 30
5.4.4. Erhalt von JMX-Benachrichtigungen 31
5.4.5. Zugriff auf Cache MBeans in eigenständiger Umgebung mit Hilfe des jconsole-Dienstprogramms 32

Kapitel 6. Versionskompatibilität und Interoperabilität 34
6.1. API-Kompatibilität 34
6.2. Interoperabilität auf Protokollebene 34
6.3. Kompatibilitätsmatrix 34

Teil II. JBoss Cache Architektur 35

Kapitel 7. Architektur 36
7.1. Datenstrukturen innerhalb des Caches 36
7.2. SPI-Schnittstellen 36
7.3. Methodenaufrufe auf Knoten 37
7.3.1. Interzeptoren 38
7.3.1.1. Schreiben angepasster Interzeptoren 38
7.3.2. Commands (Befehle) und Visitors (Besucher) 38
7.3.3. InvocationContexts 39
7.4. Manager für Untersysteme 39
7.4.1. RpcManager 39
7.4.2. BuddyManager 39
7.4.3. CacheLoaderManager 39
7.5. Marshalling und Wire-Formate 40
7.5.1. Die Marshaller-Schnittstelle 40
7.5.2. VersionAwareMarshaller 40
7.6. Klassenladen und Bereiche 41

Kapitel 8. Cache-Modi und Clustering 42
8.1. Cache-Replikationsmodi 42
8.1.1. Lokaler Modus 42
8.1.2. Replizierte Caches 42
8.1.2.1. Replizierte Caches und Transaktionen 42
8.1.2.1.1. Ein-Phasen-Festschreibung 42
8.1.2.1.2. Zwei-Phasen-Festschreibung 43
8.1.2.2. Buddy-Replikation 43
8.1.2.2.1. Auswahl von Buddys 43
8.1.2.2.2. BuddyPools 44
8.1.2.2.3. Ausfallsicherheit 44
8.1.2.2.4. Konfiguration 45
8.2. Invalidierung 45
8.3. Zustandsübertragung 45
8.3.1. Arten der Zustandsübertragung 45
8.3.2. Byte-Array und Streaming-basierte Zustandsübertragung 46
8.3.3. Vollständige und teilweise Zustandsübertragung 46
8.3.4. Transiente ("in-memory" oder speicherinterne) und persistente Zustandsübertragung 47
8.3.5. Nicht-sperrende Zustandsübertragung 48
8.3.6. Konfiguration der Zustandsübertragung 48

Kapitel 9. Cache Loader 49
9.1. CacheLoader-Schnittstelle und Lebenszyklus 49
9.2. Konfiguration 50
9.2.1. Singleton-Store-Konfiguration 52
9.3. Standardmäßig enthaltene Implementierungen 53
9.3.1. Dateibasierte Cache Loader 53

9.3.2. Cache Loader, die an andere Caches delegieren 54
9.3.3. JDBCCacheLoader 54
9.3.3.1. JDBCCacheLoader-Konfiguration 55
9.3.3.1.1. Tabellenkonfiguration 55
9.3.3.1.2. Datenquelle 55
9.3.3.1.3. JDBC-Treiber 55
9.3.3.1.4. c3p0-Verbindungs-Pooling 55
9.3.3.1.5. Konfigurationsbeispiel 56
9.3.4. S3CacheLoader 57
9.3.4.1. Amazon S3 Bibliothek 58
9.3.4.2. Konfiguration 58
9.3.5. TcpDelegatingCacheLoader 59
9.3.6. Umwandlung von Cache Loadern 60
9.4. Cache-Passivierung 60
9.4.1. Cache-Loader-Verhalten mit aktivierter bzw. deaktivierter Passivierung 61
9.5. Strategien 62
9.5.1. Lokaler Cache mit Speicher 62
9.5.2. Replizierte Caches, bei dem alle Knoten denselben Speicher verwenden 62
9.5.3. Replizierte Caches, darunter nur ein Knoten mit Speicher 63
9.5.4. Replizierte Caches, bei denen jeder Knoten seinen eigenen Speicher hat 63
9.5.5. Hierarchische Caches 64
9.5.6. Mehrere Cache Loader 65

Kapitel 10. Eviction **67**
10.1. Aufbau 67
10.1.1. Sammeln statistischer Daten 67
10.1.2. Bestimmen der zu räumenden Knoten 67
10.1.3. Art und Weise der Knotenräumung 67
10.1.4. Eviction-Threads 68
10.2. Eviction-Bereiche 68
10.2.1. Residente Knoten 68
10.3. Konfiguration der Eviction 69
10.3.1. Grundlegende Konfiguration 69
10.3.2. Befehlsorientierte Konfiguration 69
10.4. Enthaltene Eviction-Richtlinien 70
10.4.1. LRUAlgorithm – Least Recently Used ("Am längsten nicht verwendet") 70
10.4.2. FIFOAlgorithm - First In, First Out ("Als Erster rein, als Erster raus") 70
10.4.3. MRUAlgorithm - Most Recently Used ("Zuletzt verwendet") 71
10.4.4. LFUAlgorithm - Least Frequently Used ("Am wenigsten verwendet") 71
10.4.5. ExpirationAlgorithm 71
10.4.6. ElementSizeAlgorithm – Eviction basierend auf der Anzahl von Schlüssel/Wert-Paaren in einem Knoten 72

Kapitel 11. Transaktionen und Nebenläufigkeit **73**
11.1. Nebenläufiger Zugriff 73
11.1.1. Multi-Version Concurrency Control (MVCC) 73
11.1.1.1. MVCC-Konzepte 73
11.1.1.2. MVCC-Implementierung 73
11.1.1.2.1. Isolationsebenen 74
11.1.1.2.2. Nebenläufige Schreibvorgänge und Write-Skews 74
11.1.1.3. Konfiguration von Sperren 75
11.1.2. Pessimistische und optimistische Sperrschemata 75
11.2. JTA-Unterstützung 75

Teil III. JBoss Cache Konfigurationsreferenzen **77**

Kapitel 12. Konfigurationsreferenzen **78**
12.1. Beispiel einer XML-Konfigurationsdatei 78
12.1.1. XML-Validierung 82
12.2. Kurzanleitung zur Konfigurationsdatei 83

Kapitel 13. JMX-Referenzen **119**
13.1. JBoss Cache Statistiken 119
13.2. JBoss MBean Benachrichtigungen 122

Vorwort

Dies ist das offizielle JBoss Cache Benutzerhandbuch. Zusammen mit den zugehörigen Dokumenten (ein FAQ, ein Tutorial und eine ganze Reihe von Dokumenten über POJO Cache) ist es frei verfügbar auf der JBoss Cache Dokumentations-Website.

JBoss Cache bezieht sich im Folgenden auf JBoss Cache Core, einen geclusterten, transaktionalen Cache in Baumstruktur. Der POJO Cache, der ebenfalls Teil der JBoss Cache Distribution ist, wird separat dokumentiert. (POJO Cache ist ein Cache, der Plain Old Java Objects einschließlich ihrer Objektbeziehungen handhabt, mit der Fähigkeit, solche POJOs zu clustern und gleichzeitig deren Beziehungen zu wahren. Weitere Informationen diesbezüglich finden Sie in der POJO Cache Dokumentation.)

Dieses Buch richtet sich an Entwickler, die JBoss Cache entweder als eigenständigen In-Memory-Cache, als verteilten oder replizierten Cache, als Clustering-Bibliothek oder als In-Memory-Datenbank einsetzen möchten. Es richtet sich an Applikationsentwickler, die JBoss Cache in Ihrer Code-Basis einsetzen möchten, wie auch an "OEM"-Entwickler, die auf JBoss Cache Features aufbauen und diese erweitern möchten. Als solches ist dieses Buch in zwei Hauptteile unterteilt – der eine behandelt die Programmierschnittstelle für Benutzer, der andere geht sehr viel detaillierter auf spezielle Themen und auf die JBoss Cache Architektur ein.

Im Allgemeinen sind gute Kenntnisse der Programmiersprache Java sowie ein umfassendes Verständnis von Transaktionen und nebenläufiger Programmierung notwendig. Es werden jedoch keine vorherigen Kenntnisse des JBoss Application Servers vorausgesetzt oder benötigt.

Nutzen Sie für weiterführende Diskussionen das Benutzerforum, das auf der JBoss Cache Website zur Verfügung steht. Außerdem bieten wir ein Verfahren zur Nachverfolgung von Fehlermeldungen und Funktionswünschen im JBoss Cache JIRA Issue Tracker.

Falls Sie Interesse an der Weiterentwicklung des JBoss Cache oder an der Übersetzung dieser Dokumentation in weitere Sprachen haben, würden wir uns freuen, von Ihnen zu hören. Stellen Sie dazu bitte eine Nachricht im JBoss Cache Benutzerforum ein oder kontaktieren Sie uns über die JBoss Cache Entwickler-Mailing-Liste.

Dieses Buch bezieht sich speziell auf die JBoss Cache Release derselben Versionsnummer. Es gilt möglicherweise nicht für ältere oder neuere JBoss Cache Releases. Es ist daher wichtig, dass Sie die Dokumentation entsprechend derjenigen Version des JBoss Cache verwenden, die Sie später einsetzen werden.

Ich bin immer offen für Feedback, Vorschläge und Verbesserungen; richten Sie diese bitte an die Entwickler-Mailing-Liste, nicht an einzelne Autoren. Wir hoffen, dass dieses Handbuch Ihnen gute Dienste leistet und wünschen viel Spaß beim Lesen!

Manik Surtani, Oktober 2008

Teil I. Einführung in den JBoss Cache

Dieser Abschnitt behandelt alles, was ein Entwickler für einen schnellen Einsatz von JBoss Cache in seinen Projekten benötigt. Er enthält eine Übersicht über die zugrunde liegenden Konzepte, sowie Informationen über APIs, Konfiguration und Deployment.

Kapitel 1. Überblick

1.1. Was ist JBoss Cache?

JBoss Cache ist ein geclusterter, transaktionaler Cache in Baumstruktur. Er kann in einer eigenständigen, nicht-geclusterten Umgebung eingesetzt werden, um häufig verwendete Daten im Hauptspeicher zwischenzuspeichern und somit Engpässe beim Datenabruf oder bei der Verarbeitung zu vermeiden und gleichzeitig Funktionalitäten der "Unternehmensklasse" wie z.B. JTA-Kompatibilität, Eviction und Persistenz zu bieten.

JBoss Cache ist zudem ein geclusterter Cache und kann in einem Cluster dazu verwendet werden, Zustände zu replizieren, um somit eine hohe Ausfallsicherheit zu bieten. Eine Vielzahl an Replikationsmodi werden unterstützt, einschließlich Invalidierung und Buddy-Replikation, und Netzwerkkommunikation kann entweder synchron oder asynchron erfolgen.

Wird der Cache im geclusterten Modus eingesetzt, kann dadurch sehr effektiv Hochverfügbarkeit, Fehlertoleranz und Lastverteilung in benutzerdefinierte Applikationen und Frameworks integriert werden. Beispielsweise machen der JBoss Application Server und Red Hat's Enterprise Application Platform umfassend Gebrauch vom JBoss Cache, um Dienste wie HTTP und EJB-Sessions zu clustern, sowie um einen verteilten Entity-Cache für JPA bereitzustellen.

1.1.1. Und was ist ein POJO Cache?

POJO Cache ist eine Erweiterung der Kern-JBoss-Cache-API. POJO Cache bietet zusätzliche Funktionalität wie z.B.:

- bewahren von Objektreferenzen selbst nach Replikation oder Persistierung
- feingranulare Replikation, bei der nur veränderte Objektfelder repliziert werden
- "API-loses" Clustering-Modell, in dem POJOs einfach als "geclustert" annotiert werden

Für den POJO Cache steht eine eigene, umfassende Reihe von Dokumentationen, einschließlich Benutzerhandbuch, FAQ und Tutorial, auf der JBoss Cache Dokumentations-Website zur Verfügung. Daher wird auf POJO Cache in diesem Handbuch nicht näher eingegangen.

1.2. Kurzfassung der Features

1.2.1. Cachen von Objekten

JBoss Cache stellt eine einfache und unkomplizierte Programmierschnittstelle bereit, über die Daten – einfache Java-Objekte – im Cache abgelegt werden können. Basierend auf den gewählten Konfigurationsoptionen werden diese Daten (eine oder mehrere Aussagen können zutreffen):

- zwischengespeichert im Hauptspeicher für effizienten, Thread-sicheren Abruf
- repliziert über einige oder alle Instanzen in einem Cluster
- persistiert auf Festplatte und/oder auf einem entfernen, In-Memory-Cache-Cluster ("Far-Cache")
- vom Speicher bereinigt, sobald nur noch wenig Speicher verfügbar ist, und auf Festplatte passiviert, ohne dass der Zustand verloren geht

Zusätzlich bietet JBoss Cache eine breite Palette von Funktionen der Unternehmensklasse:

- die Möglichkeit, an JTA-Transaktionen teilzunehmen (funktioniert mit den meisten Java-EE-konformen Transaktionsmanagern).
- Verknüpfung mit JMX-Konsolen für Laufzeitstatistiken über den Zustand des Caches.

- Client-Code kann mit Listenern verknüpfen und Benachrichtigungen über Cache-Ereignisse erhalten.
- Cache-Operationen können in Gruppen zusammengefasst werden für effiziente Replikation.

1.2.2. Lokale und geclusterte Modi

Der Cache ist in einer Baumstruktur mit einer einzigen Wurzel organisiert. Jeder Knoten im Baum enthält im Wesentlichen eine Map, die als Speicher für Schlüssel/Wert-Paare dient. Zu cachende Objekte müssen als einzige Voraussetzung diese **`java.io.Serializable`** implementieren.

Ein JBoss Cache kann entweder lokal oder repliziert sein. Lokale Caches existieren nur innerhalb der Grenzen der JVM, in der sie erstellt wurden, während replizierte Caches Änderungen an einige oder alle anderen Caches im selben Cluster weitergeben. Ein Cluster kann sich über verschiedene Hosts auf einem Netzwerk erstrecken oder auch nur über verschiedene JVMs auf einem einzelnen Host.

1.2.3. Geclusterte Caches und Transaktionen

Wird an einem Objekt im Cache eine Änderung vorgenommen und erfolgt diese im Kontext einer Transaktion, dann wird mit der Replikation dieser Änderungen bis zur erfolgreichen Festschreibung der Transaktion gewartet. Alle Änderungen werden in einer mit der Transaktion des Aufrufers verknüpften Liste verwahrt. Bei Festschreibung der Transaktion werden die Änderungen repliziert. Andernfalls (beim Zurücksetzen bzw. "Rollback") werden die Änderungen einfach lokal rückgängig gemacht und die Änderungsliste verworfen, wodurch keinerlei Netzwerkverkehr oder Overhead verursacht wird. Führt ein Aufrufer beispielsweise 100 Änderungen durch und setzt die Transaktion dann zurück, so wird nichts repliziert, d.h. es wird kein Netzwerkverkehr verursacht.

Ist keine Transaktion und kein Batch mit dem Aufrufer assoziiert, werden Änderungen sofort repliziert, d.h. im obigen Beispiel würden für jede Änderung 100 Nachrichten gesendet werden. In diesem Zusammenhang kann man sich die Ausführung ohne eine Transaktion analog zur Ausführung mit aktiviertem "auto-commit" in JDBC-Terminologie vorstellen, bei der jede Operation sofort automatisch festgeschrieben wird.

JBoss Cache funktioniert standardmäßig mit den verbreitetsten Transaktionsmanagern und bietet darüberhinaus eine Programmierschnittstelle, an der angepasste Transaktionsmanager-Lookups geschrieben werden können.

Alle oben getroffenen Aussagen gelten auch für Batches, die ein ähliches Verhalten aufweisen.

1.2.4. Thread-Sicherheit

Der Cache ist vollständig Thread-sicher. Er setzt "Multi-Versioned Concurrency Control" (MVCC) ein, um die Thread-Sicherheit zwischen Lese- und Schreibvorgängen zu gewährleisten und gleichzeitig ein hohes Maß an Nebenläufigkeit zu bieten. Die in JBoss Cache verwendete spezifische MVCC-Implementierung ermöglicht lesende Threads gänzlich ohne Sperren und synchronisierte Blockierungen, wodurch Spitzenleistungen für leseintensive Applikationen erreicht werden. Sie nutzt außerdem angepasste, extrem leistungsfähige Sperrimplementierungen, die moderne "compare-and-swap"-Techniken für Schreib-Threads anwenden, die für Multi-Core-Architekturen optimiert wurden.

Multi-Versioned Concurrency Control (MVCC) ist seit JBoss Cache 3.x das standardmäßige Sperrschema. Die optimistischen und pessimistischen Sperrschemata älterer Versionen von JBoss Cache sind zwar noch verfügbar, wurden jedoch zu Gunsten von MVCC stillgelegt und werden aus zukünftigen Releases entfernt werden. Von der Verwendung dieser veralteten Sperrschemata wird dringend abgeraten.

Die JBoss Cache MVCC-Implementierung unterstützt ausschließlich die READ_COMMITTED und REPEATABLE_READ Isolationsebenen entsprechend ihrer Datenbank-Pendants. In Kapitel 11, *Transaktionen und Nebenläufigkeit* finden Sie Einzelheiten über MVCC.

1.3. Anforderungen

JBoss Cache erfordert eine mit Java 5.0 (oder neuer) kompatible virtuelle Maschine und die entsprechende Reihe von Bibliotheken und wurde entwickelt und getestet auf Sun's JDK 5.0 und JDK 6.

Zusätzlich zu Java 5.0 hängt JBoss Cache von JGroups und Apache's commons-logging ab. Alle Bibliotheken, von denen JBoss Cache abhängt, werden mitgeliefert, sowie mehrere optionale JARs für optionale Funktionen.

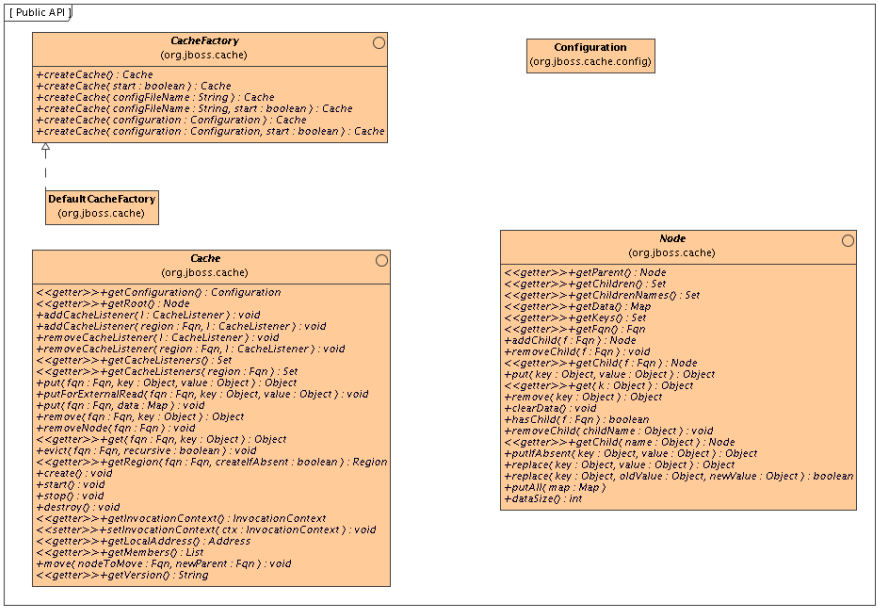
1.4. Lizenz

Bei JBoss Cache handelt es sich um ein Open-Source-Projekt, das die Unternehmens- und OEM-freundliche, OSI-anerkannte LGPL-Lizenz verwendet. Sie erhalten Unterstützung für kommerzielle Entwicklung, Produktions-Support und Training für JBoss Cache über JBoss, a division of Red Hat Inc.

Kapitel 2. Benutzer-API

2.1. API-Klassen

Die **`Cache`**-Schnittstelle ist der wichtigste Weg zur Interaktion mit JBoss Cache. Sie wird konstruiert und optional gestartet mit Hilfe der **`CacheFactory`**. Die **`CacheFactory`** ermöglicht Ihnen das Erzeugen eines **`Caches`** entweder mittels eines **`Configuration`**-Objekts oder einer XML-Datei. Der Cache organisiert Daten in einer Baumstruktur, die sich aus Knoten zusammensetzt. Sobald Sie eine Referenz auf einen **`Cache`** haben, können Sie diese nutzen, um in der Baumstruktur **`Node`**-Objekte (Knoten) zu suchen oder Daten im Baum abzulegen.


[Public API]
CacheFactory
(org.jboss.cache)
+createCache() : Cache
+createCache(start : boolean) : Cache
+createCache(configFileName : String) : Cache
+createCache(configFileName : String, start : boolean) : Cache
+createCache(configuration : Configuration) : Cache
+createCache(configuration : Configuration, start : boolean) : Cache
Configuration
(org.jboss.cache.config)
DefaultCacheFactory
(org.jboss.cache)
Cache
(org.jboss.cache)
<<getter>>+getConfiguration() : Configuration
<<getter>>+getRoot() : Node
+addCacheListener(l : CacheListener) : void
+addCacheListener(region : Fqn, l : CacheListener) : void
+removeCacheListener(l : CacheListener) : void
+removeCacheListener(region : Fqn, l : CacheListener) : void
<<getter>>+getCacheListeners() : Set
<<getter>>+getCacheListeners(region : Fqn) : Set
+put(fqn : Fqn, key : Object, value : Object) : Object
+putForExternalRead(fqn : Fqn, key : Object, value : Object) : void
+put(fqn : Fqn, data : Map) : void
+remove(fqn : Fqn, key : Object) : Object
+removeNode(fqn : Fqn) : void
<<getter>>+get(fqn : Fqn, key : Object) : Object
+evict(fqn : Fqn, recursive : boolean) : void
<<getter>>+getRegion(fqn : Fqn, createIfAbsent : boolean) : Region
+create() : void
+start() : void
+stop() : void
+destroy() : void
<<getter>>+getInvocationContext() : InvocationContext
<<setter>>+setInvocationContext(ctx : InvocationContext) : void
<<getter>>+getLocalAddress() : Address
<<getter>>+getMembers() : List
+move(nodeToMove : Fqn, newParent : Fqn) : void
<<getter>>+getVersion() : String
Node
(org.jboss.cache)
<<getter>>+getParent() : Node
<<getter>>+getChildren() : Set
<<getter>>+getChildrenNames() : Set
<<getter>>+getData() : Map
<<getter>>+getKeys() : Set
<<getter>>+getFqn() : Fqn
+addChild(f : Fqn) : Node
+removeChild(f : Fqn) : void
<<getter>>+getChild(f : Fqn) : Node
+put(key : Object, value : Object) : Object
<<getter>>+get(k : Object) : Object
+remove(key : Object) : Object
+clearData() : void
+hasChild(f : Fqn) : boolean
+removeChild(childName : Object) : void
<<getter>>+getChild(name : Object) : Node
+putIfAbsent(key : Object, value : Object) : Object
+replace(key : Object, value : Object) : Object
+replace(key : Object, oldValue : Object, newValue : Object) : boolean
+putAll(map : Map)
+dataSize() : int

Beachten Sie, dass das oben gezeigte Diagramm nur einige der häufiger verwendeten API-Methoden darstellt. Um die API zu lernen, ist der beste Weg das Lesen der Javadocs für die oben gezeigten Schnittstellen. Nachfolgend behandeln wir einige wichtige Aspekte.

2.2. Instanziierung und Start des Caches

Eine Instanz der **`Cache`**-Schnittstelle kann nur mit Hilfe einer **`CacheFactory`** erzeugt werden. Darin unterscheidet sie sich von JBoss Cache 1.x, wo eine Instanz der alten **`TreeCache`**-Klasse direkt instanziiert werden konnte.

Die **`CacheFactory`** stellt eine Reihe überladener Methoden zum Erzeugen eines **`Cache`**s bereit, die jedoch alle im Wesentlichen dasselbe tun:

- Sie erlangen Zugriff auf eine **`Configuration`**, entweder indem diese mittels Methodenparameter übergeben wird oder indem XML-Inhalt analysiert wird und eine Konfiguration konstruiert wird. Der XML-Inhalt kann aus einem gegebenen Eingangs-Stream stammen, von einem Klassenpfad oder einem Ort im Dateisystem. Siehe Kapitel 3, *Konfiguration* für weitere Informationen über das Beziehen einer **`Configuration`**.
- Sie instanziieren den **`Cache`** und stellen diesem eine Referenz zur **`Configuration`** bereit.
- Optional rufen sie die **`create()`** und **`start()`**-Methoden des Caches auf.

Sehen Sie hier ein Beispiel für das einfachste Verfahren, einen Cache unter Verwendung der standardmäßigen Konfigurationswerte zu erzeugen und zu starten:

```
   CacheFactory factory = new DefaultCacheFactory();
   Cache cache = factory.createCache();
```

In diesem Beispiel fordern wir die **`CacheFactory`** dazu auf, eine Konfigurationsdatei im Klassenpfad zu suchen und zu analysieren:

```
   CacheFactory factory = new DefaultCacheFactory();
   Cache cache = factory.createCache("cache-configuration.xml");
```

In diesem Beispiel konfigurieren wir den Cache mit Hilfe einer Datei, möchten jedoch ein Konfigurationselement befehlsorientiert ändern. Also weisen wir die Factory an, den Cache nicht zu starten, sondern tun dies stattdessen selbst:

```
   CacheFactory factory = new DefaultCacheFactory();
   Cache cache = factory.createCache("/opt/configurations/cache-configuration.xml",
false);
   Configuration config = cache.getConfiguration();
   config.setClusterName(this.getClusterName());

   // Have to create and start cache before using it
   cache.create();
   cache.start();
```

2.3. Cachen und Abrufen von Daten

Lassen Sie uns als Nächstes die **`Cache`**-API verwenden, um auf einen **`Node`** im Cache zuzugreifen und ein paar einfache Lese- und Schreibvorgänge auf diesem Knoten auszuführen.

```
// Let's get a hold of the root node.
Node rootNode = cache.getRoot();

// Remember, JBoss Cache stores data in a tree structure.
// All nodes in the tree structure are identified by Fqn objects.
Fqn peterGriffinFqn = Fqn.fromString("/griffin/peter");

// Create a new Node
Node peterGriffin = rootNode.addChild(peterGriffinFqn);

// let's store some data in the node
peterGriffin.put("isCartoonCharacter", Boolean.TRUE);
peterGriffin.put("favoriteDrink", new Beer());

// some tests (just assume this code is in a JUnit test case)
assertTrue(peterGriffin.get("isCartoonCharacter"));
assertEquals(peterGriffinFqn, peterGriffin.getFqn());
assertTrue(rootNode.hasChild(peterGriffinFqn));

Set keys = new HashSet();
keys.add("isCartoonCharacter");
keys.add("favoriteDrink");

assertEquals(keys, peterGriffin.getKeys());

// let's remove some data from the node
peterGriffin.remove("favoriteDrink");

assertNull(peterGriffin.get("favoriteDrink");

// let's remove the node altogether
rootNode.removeChild(peterGriffinFqn);

assertFalse(rootNode.hasChild(peterGriffinFqn));
```

Die **`Cache`**-Schnittstelle legt außerdem put/get/remove-Operationen offen, die ganz einfach einen Fqn (siehe Abschnitt 2.4, „Die **`Fqn`**-Klasse“) als Argument akzeptieren:

```
Fqn peterGriffinFqn = Fqn.fromString("/griffin/peter");

cache.put(peterGriffinFqn, "isCartoonCharacter", Boolean.TRUE);
cache.put(peterGriffinFqn, "favoriteDrink", new Beer());

assertTrue(peterGriffin.get(peterGriffinFqn, "isCartoonCharacter"));
assertTrue(cache.getRootNode().hasChild(peterGriffinFqn));

cache.remove(peterGriffinFqn, "favoriteDrink");

assertNull(cache.get(peterGriffinFqn, "favoriteDrink");

cache.removeNode(peterGriffinFqn);

assertFalse(cache.getRootNode().hasChild(peterGriffinFqn));
```

2.3.1. Organisieren Ihrer Daten und verwenden der Knotenstruktur

Betrachten Sie einen Knoten als eine benannte, logische Gruppe von Daten. Ein Knoten sollte verwendet werden, um die Daten eines einzelnen Datensatzes zu speichern, z.B. Informationen über eine

bestimmte Person oder einen bestimmten Account. Sie sollten sich immer vor Augen führen, dass alle Aspekte des Cachens – Sperren, Cache Loading, Replikation und Eviction – auf Basis einzelner Knoten stattfinden. Infolgedessen wird alles, was durch Speicherung in einem einzelnen Knoten zusammengruppiert ist, auch als eine einzelne, atomische Einheit behandelt.

2.4. Die Fqn-Klasse

Im vorangegangenen Abschnitt wurde die **`Fqn`**-Klasse schon in den Beispielen verwendet, lassen Sie uns nun näher auf diese Klasse eingehen.

Ein vollqualifizierter Name ("fully qualified name" oder Fqn) umfasst eine Liste mit Namen, die einen Pfad zu einem bestimmten Speicherort in der Baumstruktur des Caches darstellen. Die Elemente in der Liste sind normalerweise **`Strings`**, können grundsätzlich aber jede Art von **`Objekt`** oder eine Mischung verschiedener Typen sein.

Dieser Pfad kann absolut sein (d.h. relativ zum Root-Knoten), oder relativ zu einem beliebigen Knoten im Cache. Der Dokumentation für jeden API-Aufruf, der **`Fqn`** verwendet, können Sie entnehmen, ob die API einen relativen oder absoluten **`Fqn`** erwartet.

Die **`Fqn`**-Klasse liefert eine Fülle von Factory-Methoden; siehe Javadoc für alle Möglichkeiten. Sehen Sie im Folgenden die am häufigsten verwendeten Herangehensweisen zur Erzeugung eines Fqn:

```
   // Create an Fqn pointing to node 'Joe' under parent node 'Smith'
   // under the 'people' section of the tree

   // Parse it from a String
   Fqn abc = Fqn.fromString("/people/Smith/Joe/");

   // Here we want to use types other than String
   Fqn acctFqn = Fqn.fromElements("accounts", "NY", new Integer(12345));
```

Beachten Sie, dass

```
Fqn f = Fqn.fromElements("a", "b", "c");
```

dasselbe ist wie

```
Fqn f = Fqn.fromString("/a/b/c");
```

2.5. Stoppen und Zerstören des Caches

Es empfiehlt sich, Ihren Cache zu stoppen und zu zerstören, sobald Sie ihn nicht mehr brauchen, insbesondere, wenn es sich um einen geclusterten Cache handelt und er daher einen JGroups Channel benutzt hat. Durch das Stoppen und Zerstören des Caches wird gewährleistet, dass Ressourcen wie Netzwerk-Sockets und Wartungs-Threads vollständig bereinigt werden.

```
   cache.stop();
   cache.destroy();
```

Beachten Sie, dass ein Cache, auf dem **`stop()`** aufgerufen wurde, mit einem Aufruf an **`start()`** erneut gestartet werden kann. Ebenso kann ein Cache, auf dem **`destroy()`** aufgerufen wurde, mit einem Aufruf an **`create()`** erneut erzeugt werden (und anschließend mit einem **`start()`**-Aufruf wieder gestartet werden).

2.6. Cache-Modi

Obwohl streng genommen nicht Teil der API, beeinflusst der *Modus*, in dem der Cache ausgeführt wird, doch das clusterweite Verhalten jeglicher **`put`** oder **`remove`**-Operationen, weshalb wir an dieser Stelle kurz auf die verschiedenen Modi eingehen wollen.

JBoss Cache-Modi werden mit der **`org.jboss.cache.config.Configuration.CacheMode`**-Enumeration bezeichnet. Sie umfassen:

- *LOCAL* – lokaler, nicht geclusterter Cache. Lokale Caches treten keinem Cluster bei und kommunizieren nicht mit anderen Caches in einem Cluster.
- *REPL_SYNC* – synchrone Replikation. Replizierte Caches replizieren alle Änderungen an alle anderen Caches im Cluster. Synchrone Replikation bedeutet, dass Änderungen repliziert werden und der Aufrufer sperrt, bis er Bestätigungen der erfolgten Replikation empfängt.
- *REPL_ASYNC* – asynchrone Replikation. Replizierte Caches replizieren, genau wie bei REPL_SYNC oben, alle Änderungen an alle anderen Caches im Cluster. Asynchron bedeutet, dass der Aufrufer nicht sperrt, bis er Bestätigungen der erfolgten Replikation empfängt.
- *INVALIDATION_SYNC* – Wird ein Cache für Invalidierung statt Replikation konfiguriert, erhalten bei jeder Änderung der Daten in einem Cache die anderen Caches im Cluster eine Nachricht, die sie darüber informiert, das ihre Daten nun veraltet sind und aus dem Speicher verworfen werden sollten. Dies verringert den Overhead für die Replikation, ermöglicht aber nach wie vor das Invalidieren veralteter Daten auf entfernten Caches.
- *INVALIDATION_ASYNC* – wie oben, nur dass mit diesem Invalidierungs-Modus die Invalidierungsnachrichten asynchron versendet werden.

Siehe Kapitel 8, *Cache-Modi und Clustering* für weitere Informationen über den Einfluss der Cache-Modi auf das Verhalten. Siehe Kapitel 3, *Konfiguration* für Informationen darüber, wie Cache-Modi etc. konfiguriert werden können.

2.7. Hinzufügen eines Cache Listeners – Registrieren für Cache-Ereignisse

JBoss Cache bietet ein bequemes Verfahren zum Registrieren für Benachrichtigungen über Cache-Ereignisse.

```
Object myListener = new MyCacheListener();
cache.addCacheListener(myListener);
```

Ähnliche Methoden existieren zum Entfernen oder zum Abrufen von registrierten Listenern. Werfen Sie einen Blick auf die Javadocs über die **`Cache`**-Schnittstelle für weitere Einzelheiten.

Grundsätzlich kann jede öffentliche Klasse als Listener benutzt werden, vorausgesetzt, sie ist mit der **`@CacheListener`**-Annotation annotiert. Außerdem muss die Klasse eine oder mehrere Methoden mit einer der Annotationen auf Methodenebene (im **`org.jboss.cache.notifications.annotation`**-Paket) annotiert haben. Derart annotierte Methoden müssen öffentlich sein, einen leeren Rückgabetyp haben und einen einzelnen Parameter vom Typ **`org.jboss.cache.notifications.event.Event`** oder einen seiner Untertypen akzeptieren.

- **`@CacheStarted`** – derart annotierte Methoden erhalten eine Benachrichtigung, wenn der Cache gestartet wird. Methoden müssen einen Parametertyp akzeptieren, der von **`CacheStartedEvent`** zuweisbar ist.

- **@CacheStopped** – derart annotierte Methoden erhalten eine Benachrichtigung, wenn der Cache gestoppt wird. Methoden müssen einen Parametertyp akzeptieren, der von **CacheStoppedEvent** zuweisbar ist.
- **@NodeCreated** – derart annotierte Methoden erhalten eine Benachrichtigung, wenn ein Knoten erzeugt wird. Methoden müssen einen Parametertyp akzeptieren, der von **NodeCreatedEvent** zuweisbar ist.
- **@NodeRemoved** – derart annotierte Methoden erhalten eine Benachrichtigung, wenn ein Knoten entfernt wird. Methoden müssen einen Parametertyp akzeptieren, der von **NodeRemovedEvent** zuweisbar ist.
- **@NodeModified** – derart annotierte Methoden erhalten eine Benachrichtigung, wenn ein Knoten verändert wird. Methoden müssen einen Parametertyp akzeptieren, der von **NodeModifiedEvent** zuweisbar ist.
- **@NodeMoved** – derart annotierte Methoden erhalten eine Benachrichtigung, wenn ein Knoten verschoben wird. Methoden müssen einen Parametertyp akzeptieren, der von **NodeMovedEvent** zuweisbar ist.
- **@NodeVisited** – derart annotierte Methoden erhalten eine Benachrichtigung, wenn ein Knoten gestartet wird. Methoden müssen einen Parametertyp akzeptieren, der von **NodeVisitedEvent** zuweisbar ist.
- **@NodeLoaded** – derart annotierte Methoden erhalten eine Benachrichtigung, wenn ein Knoten von einem **CacheLoader** geladen wird. Methoden müssen einen Parametertyp akzeptieren, der von **NodeLoadedEvent** zuweisbar ist.
- **@NodeEvicted** – derart annotierte Methoden erhalten eine Benachrichtigung, wenn ein Knoten aus dem Speicher verworfen wird. Methoden müssen einen Parametertyp akzeptieren, der von **NodeEvictedEvent** zuweisbar ist.
- **@NodeInvalidated** – derart annotierte Methoden erhalten eine Benachrichtigung, wenn ein Knoten aus dem Speicher aufgrund eines entfernten Invalidierungsereignisses verworfen wird. Methoden müssen einen Parametertyp akzeptieren, der von **NodeInvalidatedEvent** zuweisbar ist.
- **@NodeActivated** – derart annotierte Methoden erhalten eine Benachrichtigung, wenn ein Knoten aktiviert wird. Methoden müssen einen Parametertyp akzeptieren, der von **NodeActivatedEvent** zuweisbar ist.
- **@NodePassivated** – derart annotierte Methoden erhalten eine Benachrichtigung, wenn ein Knoten passiviert wird. Methoden müssen einen Parametertyp akzeptieren, der von **NodePassivatedEvent** zuweisbar ist.
- **@TransactionRegistered** – derart annotierte Methoden erhalten eine Benachrichtigung, wenn der Cache eine **javax.transaction.Synchronization** mit einem registrierten Transaktionsmanager registriert. Methoden müssen einen Parametertyp akzeptieren, der von **TransactionRegisteredEvent** zuweisbar ist.
- **@TransactionCompleted** – derart annotierte Methoden erhalten eine Benachrichtigung, wenn der Cache einen Aufruf zur Festschreibung oder zum Zurücksetzen von einem registrierten Transaktionsmanager erhält. Methoden müssen einen Parametertyp akzeptieren, der von **TransactionCompletedEvent** zuweisbar ist.
- **@ViewChanged** – derart annotierte Methoden erhalten eine Benachrichtigung, wenn sich die Gruppenstruktur des Clusters ändert. Methoden müssen einen Parametertyp akzeptieren, der von **ViewChangedEvent** zuweisbar ist.
- **@CacheBlocked** – derart annotierte Methoden erhalten eine Benachrichtigung, wenn der Cluster anfordert, Cache-Operationen für ein Zustandsübertragungsereignis zu sperren. Methoden müssen einen Parametertyp akzeptieren, der von **CacheBlockedEvent** zuweisbar ist.

- **`@CacheUnblocked`** – derart annotierte Methoden erhalten eine Benachrichtigung, wenn der Cluster anfordert, Cache-Operationen nach einem Zustandsübertragungsereignis wieder freizugeben. Methoden müssen einen Parametertyp akzeptieren, der von **`CacheUnblockedEvent`** zuweisbar ist.
- **`@BuddyGroupChanged`** – derart annotierte Methoden erhalten eine Benachrichtigung, wenn ein Knoten seine Buddy-Gruppe wechselt, z.B. wenn ein Buddy aus dem Cluster ausscheidet oder ein neuer, näherer Buddy eintritt. Methoden müssen einen Parametertyp akzeptieren, der von **`BuddyGroupChangedEvent`** zuweisbar ist.

In den Javadocs über die Annotationen und über den **`Event`**-Untertyp finden Sie detaillierte Informationen darüber, was in Ihre Methode übergeben wird und wann.

Beispiel:

```
@CacheListener
public class MyListener
{

   @CacheStarted
   @CacheStopped
   public void cacheStartStopEvent(Event e)
   {
      switch (e.getType())
      {
         case CACHE_STARTED:
            System.out.println("Cache has started");
            break;
         case CACHE_STOPPED:
            System.out.println("Cache has stopped");
            break;
      }
   }

   @NodeCreated
   @NodeRemoved
   @NodeVisited
   @NodeModified
   @NodeMoved
   public void logNodeEvent(NodeEvent ne)
   {
      log("An event on node " + ne.getFqn() + " has occured");
   }
}
```

2.7.1. Synchrone und asynchrone Benachrichtigungen

Standardmäßig sind alle Benachrichtigungen insofern synchron, als diese auf dem Thread des Aufrufers erfolgen, der das Ereignis generiert hat. Infolgedessen ist es ratsam sicherzustellen, dass Cache-Listener-Implementierungen den Thread nicht in langlaufenden Aufgaben festhalten. Alternativ können Sie das **`CacheListener.sync`**-Attribut auf **`false`** setzen, so dass Sie nicht im aufrufenden Thread benachrichtigt werden. Siehe Tabelle 12.13, „Das **`<listeners />`**-Element“ für Informationen über das Optimieren dieses Thread-Pools und der Größe der Sperr-Queue.

2.8. Verwenden von Cache Loadern

Cache Loader sind ein wichtiger Teil von JBoss Cache. Sie ermöglichen das Persistieren von Knoten auf

Festplatte oder auf entfernte Cache-Cluster, und ermöglichen das Passivieren, wenn Caches nicht mehr genügend Speicherplatz haben. Darüber hinaus erlauben Cache Loader es JBoss Cache, sog. "Warmstarts" durchzuführen, wobei speicherinterne Zustände aus persistentem Speicher vorgeladen werden. JBoss Cache wird mit einer Reihe von Cache-Loader-Implementierungen geliefert.

- **`org.jboss.cache.loader.FileCacheLoader`** – ein einfacher, dateisystembasierter Cache Loader, der Daten auf Festplatte persistiert. Nicht transaktional und nicht sehr leistungsfähig, aber eine sehr einfache Lösung. Wird hauptsächlich eingesetzt zum Testen und ist nicht zum Produktionseinsatz empfohlen.
- **`org.jboss.cache.loader.JDBCCacheLoader`** – nutzt eine JDBC-Verbindung zum Speichern von Daten. Verbindungen sollten in einem internen Pool (verwendet die c3p0 Pooling-Bibliothek) erzeugt und verwaltet werden oder von einer konfigurierten Datenquelle. Die Datenbank, mit der dieser Cache Loader verbindet, kann entweder lokal oder entfernt sein.
- **`org.jboss.cache.loader.BdbjeCacheLoader`** – nutzt Oracle's BerkeleyDB dateibasierte, transaktionale Datenbank zum Persistieren von Daten. Transaktional und sehr performant, aber eine möglicherweise einschränkende Lizenz.
- **`org.jboss.cache.loader.JdbmCacheLoader`** – eine Open-Source-Alternative zur BerkeleyDB.
- **`org.jboss.cache.loader.tcp.TcpCacheLoader`** – nutzt einen TCP-Socket zum Persistieren von Daten in einem entfernten Cluster, unter Verwendung eines "Far Cache"-Musters.
- **`org.jboss.cache.loader.ClusteredCacheLoader`** – wird als schreibgeschützter Cache verwendet, wobei von anderen Knoten im Cluster der Zustand abgerufen wird. Nützlich, wenn vollständige Zustandsreplikation zu ressourcenintensiv ist und der Zustand nur bei Bedarf geladen werden soll.

Diese Cache Loader werden, zusammen mit weiterführenden Aspekten und Optimierungsmöglichkeiten, in Kapitel 9, *Cache Loader* behandelt.

2.9. Verwenden von Eviction-Richtlinien

Eviction-Richtlinien sind das Gegenstück zu Cache Loadern. Sie sind nötig um sicherzustellen, dass der Cache über genügend Speicherplatz verfügt. Sobald der Speicherplatz knapp wird, wird in einem separaten Thread ein Eviction-Algorithmus ausgeführt, der Zustände aus dem Speicher verweist und so Speicherplatz freimacht. Falls zusammen mit einem Cache Loader konfiguriert, können diese Zustände anschließend bei Bedarf vom Cache Loader abgerufen werden.

Eviction-Richtlinien können pro Region konfiguriert werden, so dass verschiedene Unterbäume im Cache über verschiedene Eviction-Einstellungen verfügen können. JBoss Cache wird mit mehreren Eviction-Richtllinien geliefert:

- **`org.jboss.cache.eviction.LRUPolicy`** – Eine Eviction-Richtlinie, die bei Erreichen des Schwellenwerts die am längsten nicht verwendeten Knoten verwirft.
- **`org.jboss.cache.eviction.LFUPolicy`** – Eine Eviction-Richtlinie, die bei Erreichen des Schwellenwerts die am wenigsten verwendeten Knoten verwirft.
- **`org.jboss.cache.eviction.MRUPolicy`** – Eine Eviction-Richtlinie, die bei Erreichen des Schwellenwerts die zuletzt verwendeten Knoten verwirft.
- **`org.jboss.cache.eviction.FIFOPolicy`** – Eine Eviction-Richtlinie, die eine First-in-First-out-Warteschlange erzeugt und bei Erreichen des Schwellenwerts die ältesten Knoten verwirft.
- **`org.jboss.cache.eviction.ExpirationPolicy`** – Eine Eviction-Richtlinie, die zu verwerfende Knoten auf Grundlage einer Verfallszeit, die für jeden Knoten konfiguriert ist, auswählt.
- **`org.jboss.cache.eviction.ElementSizePolicy`** – Eine Eviction-Richtlinie, die zu

verwerfende Knoten auf Grundlage von im Knoten enthaltenen Schlüssel/Wert-Paaren auswählt.

Detaillierte Informationen über die Konfiguration und über das Implementieren von angepassten Eviction-Richtlinien finden Sie in Kapitel 10, *Eviction*.

Kapitel 3. Konfiguration

3.1. Konfigurationsüberblick

Die **`org.jboss.cache.config.Configuration`**-Klasse (zusammen mit ihren Konfigurationselementen, siehe Abschnitt 3.3, „Zusammensetzung eines **`Configuration`**-Objekts“) ist eine Java Bean, die die Konfiguration des **`Caches`** und aller zugehörigen Architektur-Elemente (Cache Loader, Eviction-Richtlinien, etc.) umfasst.

Die **`Configuration`**-Klasse legt zahlreiche Eigenschaften offen, die im Abschnitt 12.2, „Kurzanleitung zur Konfigurationsdatei“ in diesem Buch zusammengefasst werden, und von denen viele in späteren Kapiteln wieder aufgegriffen werden. Jedes Mal, wenn in diesem Handbuch von einer Konfigurationsoption die Rede ist, können Sie davon ausgehen, dass die **`Configuration`**-Klasse oder einer ihrer Bestandteile eine einfache Eigenschafts-Setter/Getter für diese Konfigurationsoption offenlegt.

3.2. Erzeugen einer `Konfiguration`

Wie in Abschnitt 2.2, „Instanziierung und Start des Caches“ erläutert, muss der **`CacheFactory`** ein **`Configuration`**-Objekt, ein Dateiname oder ein Eingabe-Stream zur Analyse einer **`Configuration`** von XML zur Verfügung gestellt werden, bevor ein **`Cache`** erzeugt werden kann. Die folgenden Abschnitte beschreiben das nötige Vorgehen.

3.2.1. Analyse einer XML-basierten Konfigurationsdatei

Der bequemste Weg zur Konfiguration des JBoss Cache ist mittels einer XML-Datei. Die JBoss Cache Distribution enthält eine Reihe von Konfigurationsdateien für häufige Anwendungsfälle. Wir empfehlen Ihnen, diese Dateien als Ausgangspunkt zu verwenden und entsprechend Ihren Bedürfnissen anzupassen.

Das einfachste Beispiel einer Konfigurations-XML-Datei, ein Cache zur Ausführung im LOCAL-Modus, sieht folgendermaßen aus:

```
<?xml version="1.0" encoding="UTF-8"?>

<jbosscache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="urn:jboss:jbosscache-core:config:3.1">
</jbosscache>
```

Diese Datei verwendet sinnvolle Standardwerte für die Isolationsebene, Zeitüberschreitungen für Sperren, Sperrmodi, etc. Eine andere, etwas komplexere Bespiel-XML-Datei finden Sie im Abschnitt 12.1, „Beispiel einer XML-Konfigurationsdatei“-Abschnitt dieses Handbuchs, zusammen mit Abschnitt 12.2, „Kurzanleitung zur Konfigurationsdatei“ zur Erklärung der verschiedenen Optionen.

3.2.2. Überprüfen von Konfigurationsdateien

Standardmäßig überprüft JBoss Cache Ihre XML-Konfigurationsdatei anhand eines XML-Schemas und gibt eine Ausnahme aus, falls die Konfiguration ungültig ist. Dies kann mit dem **`-Djbosscache.config.validate=false`**-JVM-Parameter außer Kraft gesetzt werden. Alternativ können Sie Ihr eigenes Schema, das zur Überprüfung herangezogen werden soll, mit Hilfe des **`-`**

`Djbosscache.config.schemaLocation=url`-Parameters spezifizieren.

Standardmäßig werden Konfigurationsdateien jedoch anhand des JBoss Cache Konfigurationsschemas überprüft, das in der **`jbosscache-core.jar`** enthalten ist oder unter **`http://www.jboss.org/jbosscache/jbosscache-config-3.0.xsd`**. Die meisten XML-Bearbeitungstools können mit diesem Schema verwendet werden um sicherzustellen, dass die von Ihnen erstellte Konfigurationsdatei fehlerfrei und gültig ist.

3.2.3. Befehlsorientierte Konfiguration

Zusätzlich zur XML-basierten Konfiguration oben kann die **`Configuration`** auch befehlsorientiert erzeugt werden, unter Verwendung der einfachen Eigenschafts-Mutators, die von **`Configuration`** und deren Komponenten offen gelegt werden. Bei der Erzeugung des **`Configuration`**-Objekts wird dieses mit den JBoss Cache Standardwerten voreingestellt und kann für einen schnellen Start unverändert eingesetzt werden.

```
Configuration config = new Configuration();
config.setTransactionManagerLookupClass(
  GenericTransactionManagerLookup.class.getName() );
config.setIsolationLevel(IsolationLevel.READ_COMMITTED);
config.setCacheMode(CacheMode.LOCAL);
config.setLockAcquisitionTimeout(15000);

CacheFactory factory = new DefaultCacheFactory();
Cache cache = factory.createCache(config);
```

Selbst das recht einfache Beispiel oben bedeutet ziemlich mühseliges Programmieren, weshalb die XML-basierte Konfiguration bevorzugt eingesetzt werden sollte. Falls Ihre Applikation es jedoch erfordert, spricht nichts dagegen, die XML-basierte Konfiguration für den Großteil der Attribute zu nutzen, und dann auf das **`Configuration`**-Objekt zuzugreifen, um einigen Elementen andere Werte als die Standards zuzuweisen, einen Eviction-Bereich hinzuzufügen, etc.

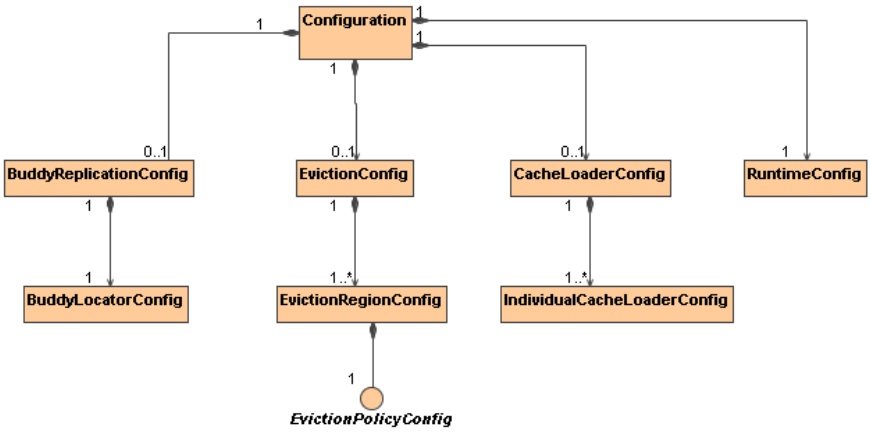
Beachten Sie, dass Konfigurationswerte nicht befehlsorientiert verändert werden dürfen, wenn der Cache läuft. Ausgenommen sind solche Werte, die als **`@Dynamic`** annotiert sind. Dynamische Eigenschaften sind in der Tabelle im Abschnitt 12.2, „Kurzanleitung zur Konfigurationsdatei“ als solches markiert. Der Versuch, eine nicht dynamische Eigenschaft zu verändern, wird eine **`ConfigurationException`** verursachen.

3.2.4. Verwenden eines IOC-Frameworks

Bei der **`Configuration`**-Klasse und ihren Konfigurationselementen (siehe Abschnitt 3.3, „Zusammensetzung eines **`Configuration`**-Objekts“) handelt es sich um Java Beans, die alle Konfigurationselemente mittels einfacher Setter und Getter offenlegen. Deshalb sollte jedes gute IOC Framework, wie z.B. JBoss Microcontainer, eine **`Configuration`** aus einer XML-Datei im eigenen Format des Frameworks erzeugen können. Siehe Abschnitt 5.2, „Via JBoss Microcontainer (JBoss AS 5.x)“ für ein Beispiel diesbezüglich.

3.3. Zusammensetzung eines `Configuration`-Objekts

Eine **`Configuration`** setzt sich aus einer Reihe von Unterobjekten zusammen:



Nachfolgend sehen Sie eine kurze Übersicht über die Komponenten einer **`Configuration`**. Siehe Javadoc und verknüpfte Kapitel dieses Handbuchs für eine ausführlichere Erklärung der mit diesen Komponenten verknüpften Konfigurationen.

- **`Configuration`** : das Objekt auf oberster Ebene der Hierarchie; legt die in Abschnitt 12.2, „Kurzanleitung zur Konfigurationsdatei“ aufgeführten Konfigurationseigenschaften offen.
- **`BuddyReplicationConfig`** : nur relevant, falls Buddy-Replikation (siehe Abschnitt 8.1.2.2, „Buddy-Replikation“) eingesetzt wird. Allgemeine Konfigurationsoptionen zur Buddy-Replikation. Muss enthalten:
- **`BuddyLocatorConfig`** : Implementierungsspezifisches Konfigurationsobjekt für die verwendete **`BuddyLocator`**-Implementierung. Welche Konfigurationselemente offengelegt werden, hängt von den Anforderungen der **`BuddyLocator`**-Implementierung ab.
- **`EvictionConfig`** : nur relevant, falls Eviction (siehe Kapitel 10, *Eviction*) eingesetzt wird. Allgemeine Eviction-Konfigurationsoptionen. Muss enthalten mindestens eine:
- **`EvictionRegionConfig`** : eine pro Eviction-Bereich; benennt den Bereich, etc. Muss enthalten:
- **`EvictionAlgorithmConfig`** : Implementierungsspezifisches Konfigurationsobjekt für die verwendete **`EvictionAlgorithm`**-Implementierung. Welche Konfigurationselemente offengelegt werden, hängt von den Anforderungen der **`EvictionAlgorithm`**-Implementierung ab.
- **`CacheLoaderConfig`** : nur relevant, falls Cache Loader (siehe Kapitel 9, *Cache Loader*) eingesetzt werden. Allgemeine Cache-Loader-Konfigurationsoptionen. Muss enthalten mindestens eine:
- **`IndividualCacheLoaderConfig`** : Implementierungsspezifisches Konfigurationsobjekt für die verwendete **`CacheLoader`**-Implementierung. Welche Konfigurationselemente offengelegt werden, hängt von den Anforderungen der **`CacheLoader`**-Implementierung ab.
- **`RuntimeConfig`** : legt Cache-Clients bestimmte Informationen über die Laufzeitumgebung des Caches offen (z.B. Mitgliedschaft in Buddy-Replikationsgruppen, falls Buddy-Replikation (siehe Abschnitt 8.1.2.2, „Buddy-Replikation“) eingesetzt wird.) Erlaubt zudem das direkte Injizieren von benötigten externen Diensten wie einem JTA **`TransactionManager`** oder einer JGroups **`ChannelFactory`** in den Cache.

3.4. Dynamische Rekonfiguration

Ein dynamisches Ändern der Konfiguration *einiger* Optionen im laufenden Betrieb des Caches wird unterstützt, indem befehlsorientiert das **`Configuration`**-Objekt vom laufenden Cache bezogen wird und Werte geändert werden, z.B.:

```
Configuration liveConfig = cache.getConfiguration();
liveConfig.setLockAcquisitionTimeout(2000);
```

Eine vollständige Aufstellung all jener Optionen, die dynamisch geändert werden können, finden Sie im Abschnitt 12.2, „Kurzanleitung zur Konfigurationsdatei“. Sollten Sie versuchen, eine Einstellung zu ändern, die nicht dynamisch ist, wird eine **`org.jboss.cache.config.ConfigurationException`** ausgegeben.

3.4.1. Überschreiben der Konfiguration mittels der Option-API

Die Option-API ermöglicht es Ihnen, bestimmte Verhaltensweisen des Caches für einen Aufruf außer Kraft zu setzen. Dazu müssen Sie eine Instanz von **`org.jboss.cache.config.Option`** erzeugen, dann müssen Sie die außer Kraft zu setzenden Optionen im **`Option`**-Objekt einstellen und dieses anschließend an den **`InvocationContext`** übergeben, bevor Sie Ihre Methode auf dem Cache aufrufen.

Um beispielsweise beim Lesen von Daten eine Schreibsperre zu erzwingen (beim Einsatz in einer Transaktion bietet dies Semantiken vergleichbar mit SELECT FOR UPDATE in einer Datenbank)

```
// first start a transaction
cache.getInvocationContext().getOptionOverrides().setForceWriteLock(true);
Node n = cache.getNode(Fqn.fromString("/a/b/c"));
// make changes to the node
// commit transaction
```

Um beispielsweise die Replikation eines put-Aufrufs in einem REPL_SYNC-Cache zu unterdrücken:

```
Node node = cache.getChild(Fqn.fromString("/a/b/c"));
cache.getInvocationContext().getOptionOverrides().setLocalOnly(true);
node.put("localCounter", new Integer(2));
```

In den Javadocs der **`Option`**-Klasse finden Sie Einzelheiten über die verfügbaren Optionen.

Kapitel 4. Batching-API

4.1. Einführung

Die Batching-API, eingeführt in JBoss Cache 3.x, bietet ein Verfahren zur stapelweisen Replikation von Aufrufen unabhängig von JTA-Transaktionen.

Dies ist hilfreich, wenn Sie Replikationsaufrufe innerhalb eines Geltungsbereichs, der kleiner als alle laufenden JTA-Transaktionen ist, stapelweise verarbeiten möchten.

4.2. Batching-Konfiguration

Um Batching zu verwenden, müssen Sie in Ihrer Cache-Konfiguration das Aufruf-Batching aktivieren, und zwar entweder im **`Configuration`**-Objekt:

```
   Configuration.setInvocationBatchingEnabled(true);
```

oder in Ihrer XML-Datei:

```
   <invocationBatching enabled="true"/>
```

Standardmäßig ist das Aufruf-Batching deaktiviert. Beachten Sie, dass Sie *nicht* notwendigerweise einen Transaktionsmanager definiert haben müssen, um Batching zu verwenden.

4.3. Batching-API

Nachdem Sie Ihren Cache zur Verwendung von Batching konfiguriert haben, benutzen Sie es durch Aufruf von **`startBatch()`** und **`endBatch()`** auf **`Cache`**, z.B.:

```
   Cache cache = getCache();

   // not using a batch
   cache.put("/a", "key", "value"); // will replicate immediately

   // using a batch
   cache.startBatch();
   cache.put("/a", "key", "value");
   cache.put("/b", "key", "value");
   cache.put("/c", "key", "value");
   cache.endBatch(true); // This will now replicate the modifications since the
batch was started.

   cache.startBatch();
   cache.put("/a", "key", "value");
   cache.put("/b", "key", "value");
   cache.put("/c", "key", "value");
   cache.endBatch(false); // This will "discard" changes made in the batch
```

Kapitel 5. Implementierung von JBoss Cache

5.1. Eigenständiger Einsatz / Befehlsorientierte Implementierung

Beim Einsatz in einem eigenständigen Java-Programm muss lediglich der Cache instanziiert werden mit Hilfe der **`CacheFactory`** und einer **`Configuration`**-Instanz oder einer XML-Datei, wie in Abschnitt 2.2, „Instanziierung und Start des Caches“ und Abschnitt 3.2, „Erzeugen einer **`Konfiguration`**“ erläutert.

Dasselbe Verfahren kann verwendet werden, wenn eine Applikation, die in einem Applikationsserver ausgeführt wird, befehlsorientiert einen Cache implementieren möchte, statt sich auf die Implementierungsfunktion des Applikationsservers zu verlassen. Ein Beispiel hierfür wäre eine Webapplikation, die einen Cache mittels eines **`javax.servlet.ServletContextListener`** implementiert.

Nach deren Erzeugung können Sie Ihre Cache-Instanz für verschiedene Applikationskomponenten freigeben, indem Sie entweder einen IOC-Container wie den JBoss Microcontainer verwenden, oder indem Sie die Cache-Instanz an JNDI binden, oder indem Sie einfach eine statische Referenz zum Cache halten.

Falls Sie nach der Implementierung Ihres Caches eine Verwaltungsschnittstelle darauf in JMX offenlegen möchten, werfen Sie einen Blick auf Abschnitt 5.4.2, „CacheJmxWrapper beim MBeanServer registrieren“.

5.2. Via JBoss Microcontainer (JBoss AS 5.x)

Ab AS 5 unterstützt JBoss AS die Implementierung von POJO-Diensten via Implementierung einer Datei, deren Name auf **`-jboss-beans.xml`** endet. Ein POJO-Dienst ist ein Dienst, dessen Implementierung via eines "Plain Old Java Object" erfolgt, d.h. eine einfache Java-Bean, die nicht zur Implementierung besonderer Schnittstellen oder zur Erweiterung bestimmter Superklassen erforderlich ist. Ein **`Cache`** ist ein POJO-Dienst, und alle Komponenten in einer **`Configuration`** sind ebenfalls POJOs, weshalb die Implementierung eines Caches auf diese Weise nahe liegt.

Die Implementierung des Caches erfolgt mit Hilfe des JBoss Microcontainers, der das Herzstück von JBoss AS bildet. JBoss Microcontainer ist ein hochentwickeltes IOC-Framework vergleichbar mit Spring. Eine **`-jboss-beans.xml`**-Datei ist im Grunde genommen ein Deskriptor, der dem IOC-Framework mitteilt, wie die verschiedenen Beans, aus denen ein POJO-Dienst besteht, zusammengesetzt werden.

Für jede konfigurierbare Option, die durch die **`Configuration`**-Komponenten offengelegt wird, muss ein Getter/Setter in der Konfigurationsklasse definiert werden. Dies ist notwendig, damit der JBoss Microcontainer auf typische IOC-Art diese Methoden aufrufen kann, wenn die entsprechenden Eigenschaften konfiguriert wurden.

Sie sollten sicherstellen, dass die **`jbosscache-core.jar`** und **`jgroups.jar`**-Bibliotheken im **`lib`**-Verzeichnis Ihres Servers vorhanden sind. Dies ist normalerweise der Fall, wenn Sie JBoss AS in der **`all`**-Konfiguration ausführen. Beachten Sie, dass Sie gegebenenfalls notwendige optionale JARs einfügen müssen, wie z.B. **`jdbm.jar`**, abhängig von Ihrer Cache-Konfiguration.

Nachfolgend sehen Sie ein Beispiel einer **`-beans.xml`**-Datei. Im **`server/all/deploy`**-Verzeichnis einer JBoss AS 5 Installation finden Sie noch einige weitere Beispiele.

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

   <!-- First we create a Configuration object for the cache -->
   <bean name="ExampleCacheConfig"
      class="org.jboss.cache.config.Configuration">

      <!-- Externally injected services -->
      <property name="runtimeConfig">
         <bean class="org.jboss.cache.config.RuntimeConfig">
            <property name="transactionManager">
               <inject bean="jboss:service=TransactionManager"
                       property="TransactionManager"/>
            </property>
            <property name="muxChannelFactory"><inject
bean="JChannelFactory"/></property>
         </bean>
      </property>

      <property name="multiplexerStack">udp</property>

      <property name="clusterName">Example-EntityCache</property>

      <property name="isolationLevel">REPEATABLE_READ</property>

      <property name="cacheMode">REPL_SYNC</property>

      <property name="stateRetrievalTimeout">15000</property>

      <property name="syncReplTimeout">20000</property>

      <property name="lockAcquisitionTimeout">15000</property>

      <property name="exposeManagementStatistics">true</property>
   </bean>

   <!-- Factory to build the Cache. -->
   <bean name="DefaultCacheFactory" class="org.jboss.cache.DefaultCacheFactory">
      <constructor factoryClass="org.jboss.cache.DefaultCacheFactory"
                   factoryMethod="getInstance" />
   </bean>

   <!-- The cache itself -->
   <bean name="ExampleCache" class="org.jboss.cache.Cache">

      <constructor factoryMethod="createCache">
          <factory bean="DefaultCacheFactory"/>
          <parameter class="org.jboss.cache.config.Configuration"><inject
bean="ExampleCacheConfig"/></parameter>
          <parameter class="boolean">false</parameter>
      </constructor>

   </bean>

</deployment>
```

Werfen Sie einen Blick auf die JBoss Microcontainer Dokumentation für Einzelheiten über die obige Syntax. Im Wesentlichen steht jedes **bean**-Element für ein Objekt und wird zur Erzeugung einer

`Configuration` und deren Konfigurationselemente (siehe Abschnitt 3.3, „Zusammensetzung eines **`Configuration`**-Objekts“) verwendet. Die **`DefaultCacheFactory`**-Bean erzeugt den Cache, wobei konzeptionell dieselben Schritte wie in Abschnitt 2.2, „Instanziierung und Start des Caches“ beschrieben durchgeführt werden.

Interessant am obigen Beispiel ist die Verwendung des **`RuntimeConfig`**-Objekts. Externe Ressourcen wie ein **`TransactionManager`** und eine JGroups **`ChannelFactory`**, die für den Microcontainer sichtbar sind, werden in die **`RuntimeConfig`** als Abhängigkeiten injiziert. Dabei wird davon ausgegangen, dass die referenzierten Beans bereits in einem anderen Deployment-Deskriptor im AS beschrieben wurden.

5.3. Automatisches Binden an JNDI in JBoss AS

Diese Funktion steht zum Zeitpunkt der Veröffentlichung dieses Handbuchs noch nicht zur Verfügung. Sobald diese Funktion verfügbar ist, werden wir eine Wiki-Seite zu diesem Thema hinzufügen.

5.4. Informationen zur Verwaltung während der Laufzeit

JBoss Cache beinhaltet JMX MBeans zur Offenlegung von Cache-Funktionalität und der Bereitstellung von Statistiken, die zur Analyse von Cache-Operationen verwendet werden können. JBoss Cache kann Cache-Ereignisse auch als MBean-Benachrichtigungen zur Handhabung via JMX Monitoring-Tools senden.

5.4.1. JBoss Cache MBeans

JBoss Cache stellt eine MBean bereit, die beim JMX-Server Ihrer Umgebung registriert werden kann, um Zugriff auf die Cache-Instanz via JMX zu ermöglichen. Diese MBean ist der **`org.jboss.cache.jmx.CacheJmxWrapper`**. Es handelt sich dabei um eine StandardMBean, ihre Schnittstelle ist also **`org.jboss.cache.jmx.CacheJmxWrapperMBean`**. Diese MBean kann verwendet werden:

- Um eine Referenz zum zugrunde liegenden **`Cache`** zu erhalten.
- Um Lebenszyklus-Operationen wie create/start/stop/destroy auf dem zugrunde liegenden **`Cache`** aufzurufen.
- Um verschiedene Details über den aktuellen Zustand des Caches einzusehen (z.B. Anzahl an Knoten, Informationen über Sperren, etc.)
- Um zahlreiche Details über die Konfiguration des Caches einzusehen und um diejenigen Konfigurationselemente, die im laufenden Betrieb des Caches veränderbar sind, bei Bedarf abzuändern.

Weitere Informationen finden Sie im **`CacheJmxWrapperMBean`** Javadoc.

Ist ein **`CacheJmxWrapper`** registriert, bietet JBoss Cache zudem MBeans für mehrere andere interne Komponenten und Untersysteme. Diese MBeans werden zum Aufzeichnen und Offenlegen der mit den jeweiligen Untersystemen im Zusammenhang stehenden Statistiken verwendet. Sie werden hierarchisch mit dem **`CacheJmxWrapper`**-MBean assoziiert und besitzen Dienstnamen, die die Beziehung verdeutlichen. Zum Beispiel kann auf ein Replikationsinterzeptor-MBean für die **`jboss.cache:service=TomcatClusteringCache`**-Instanz durch den Dienst namens **`jboss.cache:service=TomcatClusteringCache,cache-interceptor=ReplicationInterceptor`** zugegriffen werden.

5.4.2. CacheJmxWrapper beim MBeanServer registrieren

Das beste Verfahren um sicherzustellen, dass der **`CacheJmxWrapper`** in JMX registriert ist, hängt von der Implementierungsart Ihres Caches ab.

5.4.2.1. Befehlsorientierte Registrierung mit einer Cache-Instanz

Erzeugen Sie dazu einfach Ihren **`Cache`** und übergeben diesen an den **`JmxRegistrationManager`**-Konstruktor.

```
   CacheFactory factory = new DefaultCacheFactory();
   // Build but don't start the cache
   // (although it would work OK if we started it)
   Cache cache = factory.createCache("cache-configuration.xml");

   MBeanServer server = getMBeanServer(); // however you do it
   ObjectName on = new ObjectName("jboss.cache:service=Cache");

   JmxRegistrationManager jmxManager = new JmxRegistrationManager(server, cache,
on);
   jmxManager.registerAllMBeans();

   ... use the cache

   ... on application shutdown

   jmxManager.unregisterAllMBeans();
   cache.stop();
```

5.4.2.2. Befehlsorientierte Registrierung mit einer Konfigurationsinstanz

Erzeugen Sie alternativ ein **`Configuration`**-Objekt und übergeben dies an den **`CacheJmxWrapper`**. Der Wrapper wird den **`Cache`** für Sie erzeugen.

```
   Configuration config = buildConfiguration(); // whatever it does

   CacheJmxWrapperMBean wrapper = new CacheJmxWrapper(config);
   MBeanServer server = getMBeanServer(); // however you do it
   ObjectName on = new ObjectName("jboss.cache:service=TreeCache");
   server.registerMBean(wrapper, on);

   // Call to wrapper.create() will build the Cache if one wasn't injected
   wrapper.create();
   wrapper.start();

   // Now that it's built, created and started, get the cache from the wrapper
   Cache cache = wrapper.getCache();

   ... use the cache

   ... on application shutdown

   wrapper.stop();
   wrapper.destroy();
```

5.4.2.3. JMX-basiertes Deployment in JBoss AS (JBoss AS 5.x)

Der **`CacheJmxWrapper`** ist ein POJO, der Microcontainer kann also problemlos einen erzeugen. Der Trick ist nun, ihn dazu zu bringen, Ihre Bean in JMX zu registrieren. Spezifizieren Sie dazu die

`org.jboss.aop.microcontainer.aspects.jmx.JMX`-Annotation auf der **`CacheJmxWrapper`**-Bean:

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

   <!-- First we create a Configuration object for the cache -->
   <bean name="ExampleCacheConfig"
      class="org.jboss.cache.config.Configuration">

      ... build up the Configuration

   </bean>

   <!-- Factory to build the Cache. -->
   <bean name="DefaultCacheFactory" class="org.jboss.cache.DefaultCacheFactory">
      <constructor factoryClass="org.jboss.cache.DefaultCacheFactory"
                   factoryMethod="getInstance" />
   </bean>

   <!-- The cache itself -->
   <bean name="ExampleCache" class="org.jboss.cache.CacheImpl">

      <constructor factoryMethod="createnewInstance">
          <factory bean="DefaultCacheFactory"/>
          <parameter><inject bean="ExampleCacheConfig"/></parameter>
          <parameter>false</parameter>
      </constructor>

   </bean>

   <!-- JMX Management -->
   <bean name="ExampleCacheJmxWrapper"
class="org.jboss.cache.jmx.CacheJmxWrapper">


<annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(name="jboss.cache:service
=ExampleTreeCache",

exposedInterface=org.jboss.cache.jmx.CacheJmxWrapperMBean.class,
                          registerDirectly=true)</annotation>

      <constructor>
          <parameter><inject bean="ExampleCache"/></parameter>
      </constructor>

   </bean>

</deployment>
```

Wie im Abschnitt 5.4.2, „CacheJmxWrapper beim MBeanServer registrieren“ erläutert, kann der **`CacheJmxWrapper`** das Erzeugen und Starten des **`Cache`** übernehmen, wenn er mit einer **`Configuration`** ausgestattet wurde. Dies ist das bevorzugte Verfahren mit dem Microcontainer, da es die Boilerplate XML zum Erzeugen der **`CacheFactory`** erspart.

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

   <!-- First we create a Configuration object for the cache -->
   <bean name="ExampleCacheConfig"
      class="org.jboss.cache.config.Configuration">

      ... build up the Configuration

   </bean>

   <bean name="ExampleCache" class="org.jboss.cache.jmx.CacheJmxWrapper">

      <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
        (name="jboss.cache:service=ExampleTreeCache",
        exposedInterface=org.jboss.cache.jmx.CacheJmxWrapperMBean.class,
        registerDirectly=true)</annotation>

      <constructor>
          <parameter><inject bean="ExampleCacheConfig"/></parameter>
      </constructor>

   </bean>

</deployment>
```

5.4.3. JBoss-Cache-Statistiken

JBoss Cache zeichnet in seinen Interzeptoren und verschiedenen anderen Komponenten Statistiken auf und legt diese durch eine Reihe von MBeans offen. Das Sammeln von statistischen Daten ist standardmäßig aktiviert; mit Hilfe des **`Configuration.setExposeManagementStatistics()`**-Setters kann es jedoch für eine bestimmte Cache-Instanz deaktiviert werden. Beachten Sie, dass die Mehrheit aller Statistiken durch das **`CacheMgmtInterceptor`**-MBean bereitgestellt wird, so dass in dieser Hinsicht dieser Interzeptor der wichtigste ist. Wenn Sie aus Leistungsgründen alle Statistiken deaktivieren wollen, setzen Sie **`Configuration.setExposeManagementStatistics(false)`**; so wird verhindert, dass der **`CacheMgmtInterceptor`** in den Interzeptorstapel des Caches eingefügt wird, wenn der Cache gestartet wird.

Wenn ein **`CacheJmxWrapper`** bei JMX registriert ist, stellt der Wrapper zudem sicher, dass für jeden Interzeptor und jede Komponente, der/die Statistiken offenlegt, eine MBean in JMX registriert ist.

Anmerkung

Beachten Sie, dass, falls der **`CacheJmxWrapper`** nicht in JMX registriert ist, auch die Interzeptor-MBeans nicht registriert werden. Die JBoss Cache 1.4 Releases enthielten Code, der versuchte, einen **`MBeanServer`** zu finden und bei diesem automatisch die Interzeptor-MBeans zu registrieren. Für JBoss Cache 2.x haben wir entschieden, dass diese Art der "Discovery" der JMX-Umgebung über den normalen Rahmen einer Caching-Bibliothek hinaus geht, weshalb wir diese Funktionalität nunmehr entfernt haben.

Verwaltungs-Tools können anschließend auf diese MBeans zugreifen, um die Statistiken auszuwerten. Werfen Sie einen Blick auf den Abschnitt 13.1, „JBoss Cache Statistiken“, der die via JMX verfügbaren Statistiken behandelt.

5.4.4. Erhalt von JMX-Benachrichtigungen

JBoss Cache Benutzer können einen Listener zum Empfang von Cache-Ereignissen registrieren wie im Abschnitt 2.7, „Hinzufügen eines Cache Listeners – Registrieren für Cache-Ereignisse“ beschrieben. Benutzer können alternativ die Informationsinfrastruktur der Cache-Verwaltung nutzen, um diese Ereignisse über JMX-Benachrichtigungen zu empfangen. Sie können Benachrichtigungen für Cache-Ereignisse erhalten, indem Sie ein **`NotificationListener`** für den **`CacheJmxWrapper`** registrieren.

Werfen Sie einen Blick auf den Abschnitt 13.2, „JBoss MBean Benachrichtigungen“ über JMX-Benachrichtigungen für eine Liste der Benachrichtigungen, die Sie mit Hilfe des **`CacheJmxWrapper`** empfangen können.

Nachfolgend sehen Sie ein Beispiel, wie in einer JBoss Applikationsserver-Umgebung befehlsorientiert Cache-Benachrichtigungen empfangen werden können. In diesem Beispiel verwendet der Client einen Filter, um zu bestimmen, welche Ereignisse von Interesse sind.

```
   MyListener listener = new MyListener();
   NotificationFilterSupport filter = null;

   // get reference to MBean server
   Context ic = new InitialContext();
   MBeanServerConnection server =
(MBeanServerConnection)ic.lookup("jmx/invoker/RMIAdaptor");

   // get reference to CacheMgmtInterceptor MBean
   String cache_service = "jboss.cache:service=TomcatClusteringCache";
   ObjectName mgmt_name = new ObjectName(cache_service);

   // configure a filter to only receive node created and removed events
   filter = new NotificationFilterSupport();
   filter.disableAllTypes();
   filter.enableType(CacheNotificationBroadcaster.NOTIF_NODE_CREATED);
   filter.enableType(CacheNotificationBroadcaster.NOTIF_NODE_REMOVED);

   // register the listener with a filter
   // leave the filter null to receive all cache events
   server.addNotificationListener(mgmt_name, listener, filter, null);

   // ...

   // on completion of processing, unregister the listener
   server.removeNotificationListener(mgmt_name, listener, filter, null);
```

Folgendes ist die einfache Implementierung des im vorherigen Beispiel verwendeten Benachrichtigungs-Listeners.

```
   private class MyListener implements NotificationListener, Serializable
   {
      public void handleNotification(Notification notification, Object handback)
      {
         String message = notification.getMessage();
         String type = notification.getType();
         Object userData = notification.getUserData();

         System.out.println(type + ": " + message);

         if (userData == null)
         {
            System.out.println("notification data is null");
         }
         else if (userData instanceof String)
         {
            System.out.println("notification data: " + (String) userData);
         }
         else if (userData instanceof Object[])
         {
            Object[] ud = (Object[]) userData;
            for (Object data : ud)
            {
               System.out.println("notification data: " + data.toString());
            }
         }
         else
         {
            System.out.println("notification data class: " +
userData.getClass().getName());
         }
      }
   }
```

Beachten Sie, dass die JBoss Cache Management-Implementierung nur auf Cache-Ereignisse lauscht, nachdem ein Client für den Erhalt von MBean-Benachrichtigungen registriert ist. Sobald keine Clients für Benachrichtigungen registriert sind, entfernt das MBean sich selbst als Cache-Listener.

5.4.5. Zugriff auf Cache MBeans in eigenständiger Umgebung mit Hilfe des `jconsole`-Dienstprogramms

Auf JBoss Cache MBeans kann einfach zugegriffen werden, wenn Cache-Instanzen in einem Applikationsserver laufen, der eine MBean-Serverschnittstelle wie z.B. die JBoss JMX-Konsole bereitstellt. In der Server-Dokumentation finden Sie Anweisungen dazu, wie auf MBeans, die im MBean-Container eines Servers laufen, zugegriffen wird.

Zusätzlich kann mit Hilfe des **`jconsole`**-Tools Ihres JDKs auf JBoss Cache MBeans zugegriffen werden, wenn sie in einer Umgebung außerhalb eines Servers laufen. Läuft ein eigenständiger Cache außerhalb eines Applikationsservers, können Sie wie folgt auf die MBeans des Caches zugreifen.

1. Setzen Sie die Systemeigenschaft **`-Dcom.sun.management.jmxremote`** beim Start der JVM, in der der Cache laufen soll.
2. Sobald die JVM läuft, starten Sie das **`jconsole`**-Dienstprogramm, das sich in JDK's **`/bin`**-Verzeichnis befindet.
3. Wenn das Dienstprogramm geladen ist, können Sie Ihre JVM wählen und sich damit verbinden. Die JBoss Cache MBeans sind im MBeans-Fenster verfügbar.

Beachten Sie, dass das **`jconsole`**-Dienstprogramm sich automatisch als Listener für Cache-Benachrichtigungen registriert, wenn es mit einer JVM verbunden wird, die JBoss Cache-Instanzen ausführt.

Kapitel 6. Versionskompatibilität und Interoperabilität

6.1. API-Kompatibilität

Innerhalb einer Hauptversion sollten JBoss Cache Releases kompatibel und interoperabel sein. Kompatibel in dem Sinne, dass es möglich sein sollte, eine Anwendung von einer Version zu einer anderen durch Austauschen der JARs zu aktualisieren. Interoperabel in dem Sinne, dass bei Verwendung zweier verschiedener Versionen von JBoss Cache in demselben Cluster der Austausch von Replikations- und Zustandtübertragungsnachrichten möglich sein soll. Beachten Sie jedoch, dass Interoperabilität die Verwendung derselben JGroups-Version in allen Knoten des Clusters erfordert. In den meisten Fällen kann die Version von JGroups, die von einer Version von JBoss Cache verwendet wird, aktualisiert werden.

Demnach ist JBoss Cache 2.x.x nicht API- oder binärkompatibel mit früheren 1.x.x Versionen. Andererseits ist JBoss Cache 2.1.x API- und binärkompatibel mit 2.0.x.

Wir haben uns jedoch größte Mühe gegeben, JBoss Cache 3.x sowohl binär- als auch API-kompatibel mit 2.x zu halten. Nichtsdestotrotz wird empfohlen, dass der Client-Code aktualisiert wird, um keine veralteten Methoden, Klassen und Konfigurationsdateien zu verwenden.

6.2. Interoperabilität auf Protokollebene

Es steht ein Konfigurationsparameter **`Configuration.setReplicationVersion()`** zur Verfügung, um das Protokoll zur Übertragung von Inter-Cache-Kommunikation zu steuern. Die Kommunikation kann entweder effizient mit neueren Protokollen erfolgen, oder aber kann heruntergestuft werden auf kompatible Versionen zur Kommunikation mit älteren Releases. Dieser Mechanismus ermöglicht eine Verbesserung des JBoss Cache durch Einsatz effizienterer Formate, wobei gleichzeitig ein Weg zur Wahrung der Interoperabilität bestehen bleibt.

6.3. Kompatibilitätsmatrix

Auf der JBoss Cache Website wird eine Kompatibilitätsmatrix gepflegt, die Informationen über die verschiedenen Versionen von JBoss Cache, JGroups und JBoss Application Server enthält.

Teil II. JBoss Cache Architektur

Dieser Abschnitt behandelt die JBoss Cache Architektur ausführlicher und richtet sich an Entwickler, die fortgeschrittenere Cache-Funktionen verwenden möchten, den Cache erweitern oder verbessern, Plugins schreiben, oder die ganz einfach gerne einen Blick unter die Haube werfen wollen.

Kapitel 7. Architektur

7.1. Datenstrukturen innerhalb des Caches

Ein **`Cache`** besteht aus einer Gruppe von **`Node`**-Instanzen (Knoten), die in einer Baumstruktur angeordnet sind. Jeder **`Node`** enthält eine **`Map`**, welche die zu cachenden Datenobjekte enthält. Es ist wichtig zu verstehen, dass die Struktur ein mathematischer Baum ist, kein Graph; jeder **`Knoten`** hat genau ein übergeordnetes Element, und der Wurzelknoten wird durch den konstanten, vollqualifizierten Namen bezeichnet, **`Fqn.ROOT`**.

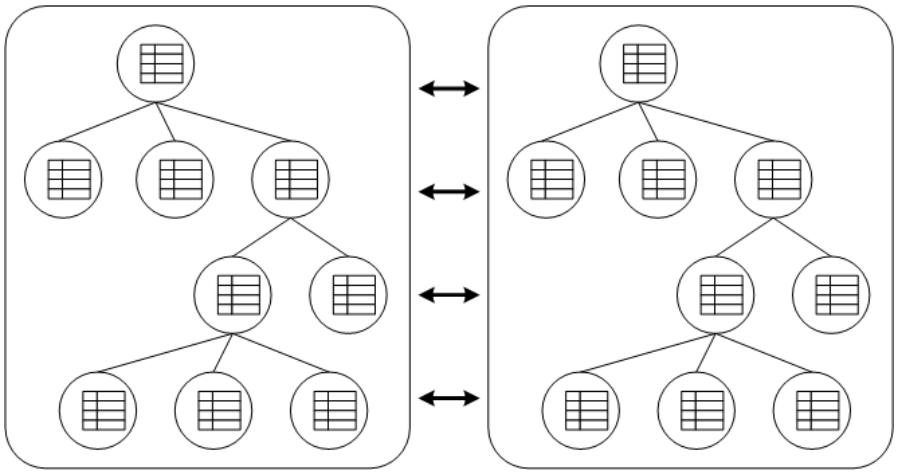


Abbildung 7.1. Daten in Baumstruktur

Im obigen Diagramm steht jeder Kasten für eine JVM. Sie sehen 2 Caches in separaten JVMs, die ihre Daten zum jeweils anderen replizieren.

Jede Änderung (siehe Kapitel 2, *Benutzer-API*) in einer Cache-Instanz wird zum anderen Cache repliziert. Selbstverständlich können Sie mehr als zwei Caches in einem Cluster haben. Abhängig von den transaktionalen Einstellungen erfolgt diese Replikation entweder nach jeder Änderung oder erst nach Abschluss einer Transaktion zum Zeitpunkt der Festschreibung. Wenn ein neuer Cache erzeugt wird, kann dieser optional beim Start die Inhalte eines anderen, vorhandenen Caches beziehen.

7.2. SPI-Schnittstellen

Zusätzlich zu den **`Cache`**- und **`Node`**-Schnittstellen legt JBoss Cache die leistungsstärkeren **`CacheSPI`**- und **`NodeSPI`**-Schnittstellen offen, die ein höheres Maß an Kontrolle über die JBoss Cache Interna erlauben. Diese Schnittstellen sind nicht zum allgemeinen Gebrauch vorgesehen, sondern richten sich an Entwickler, die den JBoss Cache erweitern oder verbessern möchten, oder angepasste **`Interceptor`**- oder **`CacheLoader`**-Instanzen schreiben möchten.

Abbildung 7.2. SPI-Schnittstellen

Die **`CacheSPI`**-Schnittstelle kann nicht erzeugt werden, sondern wird durch die **`setCache(CacheSPI cache)`**-Methoden auf diesen Schnittstellen in die **`Interceptor`**- und **`CacheLoader`**-Implementierungen injiziert. **`CacheSPI`** erweitert **`Cache`**, sämtliche Funktionen der einfachen API sind also ebenso verfügbar.

Ebenso kann auch eine **`NodeSPI`**-Schnittstelle nicht erzeugt werden. Stattdessen erhalten Sie eine, indem Sie Operationen auf **`CacheSPI`** ausführen, wie oben. Zum Beispiel wird **`Cache.getRoot() : Node`** als **`CacheSPI.getRoot() : NodeSPI`** überschrieben.

Es ist wichtig zu wissen, dass ein direktes Konvertieren eines **`Cache`** oder **`Node`** zu deren SPI-Äquivalenten nicht empfohlen wird und als schlechtes Vorgehen betrachtet wird, da die Schnittstellenvererbung kein Vertrag ist, der garantiert eingehalten wird. Die offengelegten, öffentlichen APIs andererseits sind garantiert.

7.3. Methodenaufrufe auf Knoten

Da der Cache im Grunde genommen eine Gruppe von Knoten ist, müssen Aspekte wie Clustering, Persistenz, Eviction, etc. auf diese Knoten angewendet werden, wenn Operationen auf dem Cache als Ganzes oder auf einzelnen Knoten aufgerufen werden. Um dies auf eine saubere, modulare und erweiterbare Weise zu erreichen, wird eine Interzeptorkette verwendet. Die Kette ist aus einer Reihe von Interzeptoren aufgebaut, von denen jeder einen bestimmten Aspekt oder eine bestimmte Funktionalität hinzufügt. Die Kette wird bei Erzeugung des Caches angelegt, basierend auf der verwendeten Konfiguration.

Sie sollten wissen, dass die **`NodeSPI`** einige Methoden bietet (wie z.B. die **`xxxDirect()`**-

Methodenfamilie), die direkt auf einem Knoten operieren, ohne den Interzeptorstapel zu durchlaufen. Plugin-Autoren sollten sich darüber im Klaren sein, dass die Verwendung solcher Methoden Auswirkungen auf die Aspekte des Caches hat, die ggf. angewendet werden müssen, wie z.B. Sperren, Replikation, etc. Kurzum, verwenden Sie diese Methoden nur dann, wenn Sie *wirklich* wissen, was Sie da tun!

7.3.1. Interzeptoren

JBoss Cache ist im Wesentlichen eine Kerndatenstruktur – eine Implementierung von **`DataContainer`** –, und Aspekte und Funktionen werden mit Hilfe von Interzeptoren vor dieser Datenstruktur implementiert. Ein **`CommandInterceptor`** ist eine abstrakte Klasse, Interzeptor-Implementierungen erweitern diese.

`CommandInterceptor` implementiert die **`Visitor`**-Schnittstelle, kann also Befehle auf stark typisierte Weise ändern, wenn der Befehl auf dem Weg in die Datenstruktur ist. Mehr Informationen über Visitors und Befehle finden Sie im nächsten Abschnitt.

Interzeptorimplementierungen werden in der **`InterceptorChain`**-Klasse verkettet, die einen Befehl durch die Interzeptorkette hindurch ausgibt. Ein spezieller Interzeptor, der **`CallInterceptor`**, befindet sich immer am Ende dieser Kette, um den Befehl, der durch die Kette geschickt wird, auszuführen, indem er die **`process()`**-Methode des Befehls aufruft.

JBoss Cache wird mit mehreren Interzeptoren ausgeliefert, die jeweils für verschiedene Verhaltensweisen stehen. Dazu gehören:

- **`TxInterceptor`** – sucht nach laufenden Transaktionen und meldet sich bei Transaktionsmanagern an, um an Synchronisierungsereignissen teilzunehmen
- **`ReplicationInterceptor`** – repliziert Zustände über den Cluster hinweg mit Hilfe der RpcManager-Klasse
- **`CacheLoaderInterceptor`** – lädt Daten aus einem persistenten Speicher, falls die angeforderten Daten nicht im Cache verfügbar sind

Die für Ihre Cache-Instanz konfigurierte Interzeptorkette kann durch Aufruf von **`CacheSPI.getInterceptorChain()`** abgerufen und eingesehen werden, wodurch eine geordnete **`Liste`** mit Interzeptoren in der Reihenfolge ausgegeben wird, in der sie von einem Befehl angetroffen würden.

7.3.1.1. Schreiben angepasster Interzeptoren

Sie können angepasste Interzeptoren schreiben, um bestimmte Aspekte oder Funktionen hinzuzufügen, indem Sie **`CommandInterceptor`** erweitern und die relevanten **`visitXXX()`**-Methoden überschreiben basierend auf den Befehlen, die Sie abfangen möchten. Es gibt andere, abstrakte Interzeptoren, die Sie stattdessen erweitern könnten, wie z.B. den **`PrePostProcessingCommandInterceptor`** und den **`SkipCheckChainedInterceptor`**. In den entsprechenden Javadocs finden Sie weitere Einzelheiten über die zusätzlich gebotenen Funktionen.

Der angepasste Interzeptor muss mit Hilfe der **`Cache.addInterceptor()`**-Methoden zur Interzeptorkette hinzugefügt werden. Werfen Sie für Einzelheiten einen Blick auf die Javadocs dieser Methoden.

Das Hinzufügen von angepassten Interzeptoren via XML wird ebenfalls unterstützt, siehe Kapitel 12, *Konfigurationsreferenzen* für Details.

7.3.2. Commands (Befehle) und Visitors (Besucher)

Intern nutzt JBoss Cache ein Command/Visitor-Muster zur Ausführung von API-Aufrufen. Sobald eine

Methode auf der Cache-Schnittstelle aufgerufen wird, erzeugt **`CacheInvocationDelegate`**, das die **`Cache`**-Schnittstelle implementiert, eine Instanz von **`VisitableCommand`** und leitet diesen Befehl durch die Interzeptorkette. Interzeptoren, die die **`Visitor`**-Schnittstelle implementieren, können **`VisitableCommand`**s verarbeiten, an denen sie interessiert sind, und eine Verhaltensweise zum Befehl hinzufügen.

Jeder Befehl enthält sämtliche Details über den ausgeführten Befehl, wie z.B. verwendete Parameter und Verarbeitungsverhalten, eingekapselt in einer **`process()`**-Methode. Zum Beispiel wird der **`RemoveNodeCommand`** erzeugt und durch die Interzeptorkette geschickt, wenn **`Cache.removeNode()`** aufgerufen wird, und **`RemoveNodeCommand.process()`** verfügt über notwendige Informationen darüber, wie ein Knoten aus der Datenstruktur entfernt wird.

Neben "visitable" sind Befehle auch replizierbar. Die JBoss Cache Marshaller wissen, wie Befehle effizient geordnet werden und wie sie auf entfernten Cache-Instanzen aufgerufen werden mit Hilfe eines internen RPC-Verfahrens basierend auf JGroups.

7.3.3. InvocationContexts

`InvocationContext` enthält für die Dauer eines einzelnen Aufrufs einen Zwischenzustand und wird angelegt und zerstört durch den **`InvocationContextInterceptor`**, der sich am Anfang der Interzeptorkette befindet.

`InvocationContext` enthält, wie der Name schon sagt, Kontextinformationen im Zusammenhang mit einem einzelnen Cache-Methodenaufruf. Kontextinformationen umfassen zugehörige **`javax.transaction.Transaction`** oder **`org.jboss.cache.transaction.GlobalTransaction`**, den Ursprung des Methodenaufrufs (**`InvocationContext.isOriginLocal()`**) sowie Konfigurationsoptionen (siehe Abschnitt 3.4.1, „Überschreiben der Konfiguration mittels der Option-API“), und Informationen darüber, welche Knoten gesperrt wurden, etc.

Der **`InvocationContext`** kann abgefragt werden durch Aufruf von **`Cache.getInvocationContext()`**.

7.4. Manager für Untersysteme

Manche Aspekte und Funktionen werden von mehr als einem einzelnen Interzeptor verwendet. Einige davon wurden in Manager eingekapselt, zur Verwendung durch verschiedene Interzeptoren, und werden durch die **`CacheSPI`**-Schnittstelle bereitgestellt.

7.4.1. RpcManager

Diese Klasse ist verantwortlich für Aufrufe über den JGroups-Channel für alle RPC-Aufrufe an entfernte Caches und kapselt den verwendeten JGroups-Channel ein.

7.4.2. BuddyManager

Diese Klasse verwaltet Buddy-Gruppen und tätigt entfernte Aufrufe zur Gruppenorganisation, um einen Cluster von Caches in kleinere Untergruppen aufzuteilen.

7.4.3. CacheLoaderManager

Legt Cache Loader an und konfiguriert diese. Diese Klasse umschließt einzelne **`CacheLoader`**-Instanzen in delegierenden Klassen, wie z.B. **`SingletonStoreCacheLoader`** oder **`AsyncCacheLoader`**, oder fügt den **`CacheLoader`** mit Hilfe des **`ChainingCacheLoader`** zu einer Kette hinzu.

7.5. Marshalling und Wire-Formate

Ältere Versionen von JBoss Cache schrieben gecachte Daten einfach ins Netzwerk, indem Sie während der Replikation an einen **`ObjectOutputStream`** schrieben. Über mehrere Releases in der JBoss Cache 1.x.x Serie hinweg wurde dieser Ansatz allmählich zugunsten eines ausgereifteren Marshalling-Frameworks aufgegeben. In der JBoss Cache 2.x.x Serie ist dies nun der einzig offiziell unterstützte und empfohlene Weg zum Schreiben von Objekten in Datenströme.

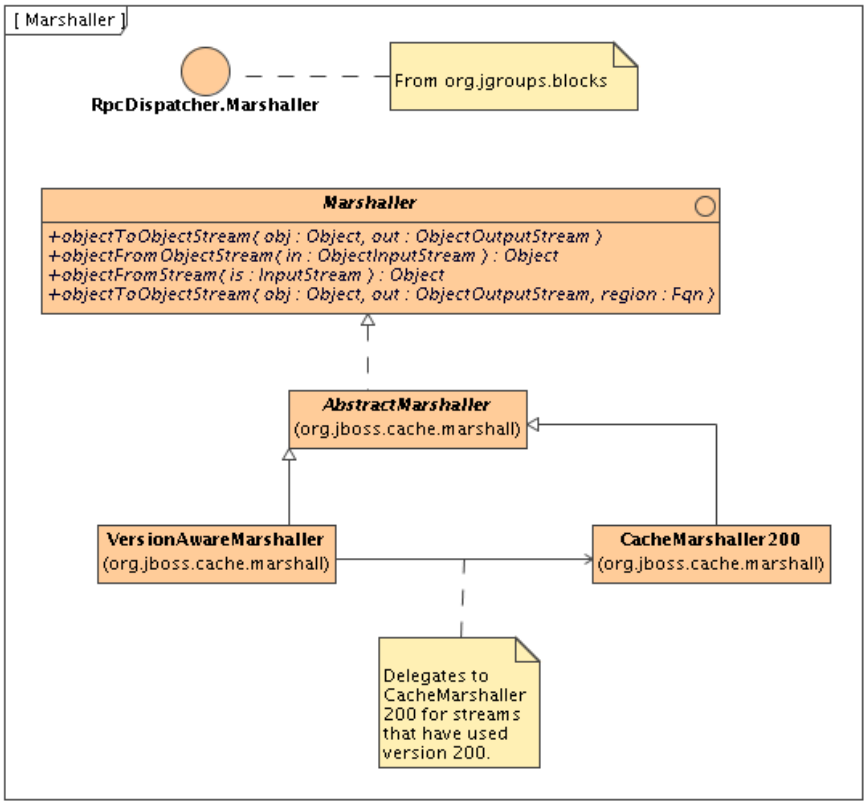


Abbildung 7.3. Die Marshaller-Schnittstelle

7.5.1. Die Marshaller-Schnittstelle

Die **`Marshaller`**-Schnittstelle erweitert **`RpcDispatcher.Marshaller`** von JGroups. Diese Schnittstelle hat zwei Hauptimplementierungen – eine delegierende **`VersionAwareMarshaller`** und eine konkrete **`CacheMarshaller300`**.

Der Marshaller kann durch Aufruf von **`CacheSPI.getMarshaller()`** bezogen werden und ist standardmäßig der **`VersionAwareMarshaller`**. Benutzer können auch ihre eigenen Marshaller schreiben, indem sie die **`Marshaller`**-Schnittstelle implementieren oder die **`AbstractMarshaller`**-Klasse erweitern, und diese in Ihre Konfiguration mit Hilfe des **`Configuration.setMarshallerClass()`**-Setters einfügen.

7.5.2. VersionAwareMarshaller

Wie der Name schon andeutet, fügt dieser Marshaller beim Schreiben einen Versions-**`Short`** am Anfang eines jeden Datenstroms hinzu, so dass ähnliche **`VersionAwareMarshaller`**-Instanzen diesen Versions-Short lesen können und dadurch wissen, an welche spezifische Marshaller-Implementierung der Aufruf zu delegieren ist. Beispielsweise ist **`CacheMarshaller200`** der Marshaller für JBoss

Cache 2.0.x. JBoss Cache 3.0.x wird mit **`CacheMarshaller300`** mit einem verbesserten Wire-Protokoll ausgeliefert. Der Einsatz eines **`VersionAwareMarshaller`** hilft dabei, die Kompatibilität von Wire-Protokollen zwischen Nebenversionen zu bewahren und erlaubt gleichzeitig die Flexibilität, das Wire-Protokoll zwischen Neben- und Microversionen zu optimieren und zu verbessern.

7.6. Klassenladen und Bereiche

Wenn die in der Applikation implementierten Applikationen zum Clustern von Zuständen von Applikationsservern verwendet werden, neigen sie dazu, Instanzen von Objekten im Cache (oder in einem **`HttpSession`**-Objekt) abzulegen, die spezifisch für ihre Applikation sind und die repliziert werden müssten. Es ist üblich, dass Applikationsserver separate **`ClassLoader`**-Instanzen für jede implementierte Applikation zuweisen, die JBoss Cache Bibliotheken jedoch vom **`ClassLoader`** des Applikationsservers referenziert werden.

Um erfolgreich Marshalling und Unmarshalling von Objekten von solchen Klassenladern durchzuführen, nutzen wir ein Konzept namens Bereiche (auch Regionen). Ein Bereich ist ein Teil des Caches, der einen gemeinsamen Klassenlader verwendet (Bereiche dienen auch anderen Zwecken, siehe Kapitel 10, *Eviction*).

Ein Bereich wird erzeugt mit Hilfe der **`Cache.getRegion(Fqn fqn, boolean createIfNotExists)`**-Methode, die eine Implementierung der **`Region`**-Schnittstelle zurückgibt. Sobald ein Bereich eingerichtet wurde, kann ein Klassenlader für diesen Bereich eingestellt werden, und der Bereich kann aktiviert bzw. deaktiviert werden. Standardmäßig sind Bereiche aktiv, es sei denn, das **`InactiveOnStartup`**-Konfigurationsattribut ist auf **`true`** gesetzt.

Kapitel 8. Cache-Modi und Clustering

Dieses Kapitel behandelt Aspekte rund um das Clustering von JBoss Cache.

8.1. Cache-Replikationsmodi

Der JBoss Cache kann entweder lokal (eigenständig) oder geclustert konfiguriert werden. Falls in einem Cluster, kann der Cache zur Replikation oder zur Invalidierung von Änderungen konfiguriert werden. Eine ausführliche Erläuterung diesbezüglich folgt.

8.1.1. Lokaler Modus

Lokale Caches schließen sich keinem Cluster an und kommunizieren nicht mit anderen Knoten in einem Cluster. Die Abhängigkeit von der JGroups-Bibliothek besteht nach wie vor, obwohl kein JGroups-Channel gestartet wird.

8.1.2. Replizierte Caches

Replizierte Caches replizieren alle Änderungen an einige oder alle anderen Cache-Instanzen im Cluster. Replikation kann entweder nach jeder Änderung stattfinden (keine Transaktionen oder Batches) oder aber nach Transaktions- bzw. Batch-Ende.

Die Replikation kann synchron oder asynchron erfolgen. Die Verwendung der jeweiligen Optionen ist abhängig von der Applikation. Synchrone Replikation sperrt den Aufrufer (z.B. nach einem **`put()`**), bis die Änderungen erfolgreich auf allen Knoten im Cluster repliziert wurden. Asynchrone Replikation führt die Replikation im Hintergrund durch (das **`put()`** kehrt sofort zurück). JBoss Cache bietet auch eine Replikationswarteschlange, wo Änderungen periodisch repliziert werden (d.h. in bestimmten Intervallen) oder wenn die Größe der Warteschlange eine bestimmte Anzahl von Elementen überschreitet, oder eine Kombination aus beidem. Eine Replikationswarteschlange bietet eine sehr viel bessere Leistung, denn die eigentliche Replikation wird von einem Thread im Hintergrund durchgeführt.

Asynchrone Replikation ist schneller (kein Sperren des Aufrufers), da synchrone Replikation Rückmeldung von allen Knoten in einem Cluster erfordert, dass diese die Änderung erfolgreich empfangen und angewendet haben (Round-Trip-Zeit). Wenn jedoch die synchrone Replikation erfolgreich zurückkehrt, so weiß der Aufrufer sicher, dass alle Änderungen auf allen Knoten angewendet wurden, während dies bei asynchroner Replikation nicht unbedingt der Fall sein muss. Bei asynchroner Replikation werden Fehler einfach in eine Protokolldatei geschrieben. Selbst bei der Verwendung von Transaktionen kann etwa eine Transaktion erfolgreich verlaufen, die Replikation aber nicht auf allen Cache-Instanzen erfolgreich sein.

8.1.2.1. Replizierte Caches und Transaktionen

Beim Einsatz von Transaktionen kommt es nur an der Transaktionsgrenze (der sog. "Transaction Boundary") zur Replikation, d.h. wenn eine Transaktion festgeschrieben wird. Dadurch wird das Ausmaß des Replikationsdatenverkehrs begrenzt, da eine einzelne Änderung übertragen wird statt einer Reihe individueller Änderungen. Dies kann wesentlich effizienter sein als das Nichtverwenden von Transaktionen. Ein weiterer Effekt ist, dass beim eventuellen Zurücksetzen einer Transaktion keinerlei Daten im Cluster übertragen werden.

Je nachdem, ob Sie Ihren Cluster im asynchronen oder synchronen Modus betreiben, verwendet JBoss Cache entweder das Ein-Phasen- oder das Zwei-Phasen-Festschreibungs-Protokoll.

8.1.2.1.1. Ein-Phasen-Festschreibung

Wird im Cache-Modus REPL_ASYNC verwendet. Alle Änderungen werden in einem einzelnen Aufruf repliziert, der entfernte Caches anweist, die Änderungen an ihrem lokalen, speicherinternen Zustand

anzuwenden und lokal festzuschreiben. Entfernte Fehler oder Zurücksetzungen werden nicht wieder an den Urheber der Transaktion zurückgeführt, da die Kommunikation asynchron ist.

8.1.2.1.2. Zwei-Phasen-Festschreibung

Wird im Cache-Modus REPL_SYNC verwendet. Nach Festschreibung Ihrer Transaktion sendet JBoss Cache einen Aufruf zur Vorbereitung ("Prepare"), der alle für die Transaktion relevanten Änderungen enthält. Entfernte Caches erhalten dann lokale Sperren auf ihrem speicherinternen Zustand und wenden die Änderungen an. Haben alle entfernten Caches auf den Aufruf zur Vorbereitung geantwortet, so sendet der Urheber der Transaktion eine Festschreibung ("Commit"). Dies weist alle entfernten Caches an, ihre Daten festzuschreiben. Antwortet auch nur einer der Caches nicht auf den Aufruf zur Vorbereitung, so sendet der Urheber einen Aufruf zum Zurücksetzen ("Rollback") der Transaktion.

Beachten Sie, dass, obwohl die "prepare"-Phase synchron ist, die "commit"- und "rollback"-Phasen asynchron sind. Der Grund hierfür ist, dass Sun's JTA-Spezifikation nicht festlegt, wie transaktionale Ressourcen an dieser Stelle der Transaktion mit Fehlschlägen umgehen sollten, und andere an der Transaktion teilhabende Ressourcen ohnehin einen unbestimmten Zustand besitzen könnten. Für diese Phase der Transaktion können wir daher auf den zusätzlichen Aufwand für synchrone Kommunikation verzichten. Die Synchronität kann jedoch mittels der **`SyncCommitPhase`** und **`SyncRollbackPhase`** Konfigurationsattribute erzwungen werden.

8.1.2.2. Buddy-Replikation

Buddy-Replikation gestattet das Unterdrücken der Replikation Ihrer Daten an alle Instanzen in einem Cluster. Stattdessen wählt jede Instanz einen oder mehrere sog. "Buddys" im Cluster und repliziert nur an diese spezifischen Buddys. Dies verbessert die Skalierbarkeit, da nicht jedesmal, wenn dem Cluster eine neue Instanz hinzugefügt wird, dies Auswirkungen auf den Speicher und Netzwerkverkehr hat.

Einer der gängigsten Anwendungsfälle der Buddy-Replikation ist die Verwendung eines replizierten Caches durch einen Servlet-Container zum Speichern von HTTP Session-Daten. Eine der Voraussetzungen, damit Buddy-Replikation gut funktioniert und auch tatsächlich von Nutzen ist, ist die *Session-Affinität*, auch als *Sticky Sessions* bekannt. Das bedeutet, dass es beim häufigen Zugriff auf bestimmte Daten besser ist, wenn der Zugriff immer auf einer bestimmten Instanz erfolgt statt reihum auf veschiedenen Instanzen. Dies hilft dem Cluster bei der Optimierung der Buddy-Auswahl, bei der Auswahl des Speicherplatzes für Daten und bei der Minimierung von Replikationsdatenverkehr.

Ist dies nicht möglich, so kann sich Buddy-Replikation eher als Nachteil denn als Vorteil erweisen.

8.1.2.2.1. Auswahl von Buddys

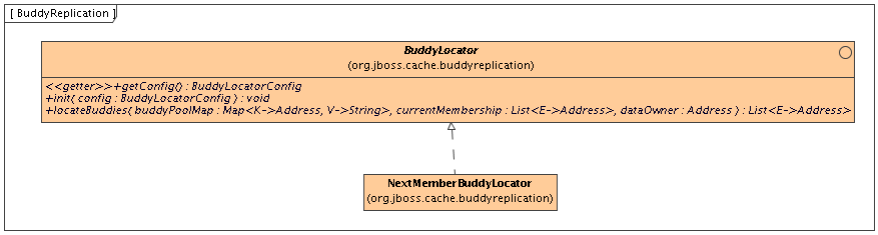


Abbildung 8.1. BuddyLocator

Buddy-Replikation verwendet eine Instanz eines **`BuddyLocator`**, der über eine Logik zur Auswahl von Buddys in einem Netzwerk verfügt. JBoss Cache wird derzeit mit einer einzelnen Implementierung geliefert, **`NextMemberBuddyLocator`**, die standardmäßig verwendet wird, wenn keine Implementierung angegeben wird. Der **`NextMemberBuddyLocator`** wählt – wie der Name bereits andeutet – das nächste Mitglied im Cluster und gewährleistet so eine gleichmäßige Verteilung für jede Instanz.

Der **`NextMemberBuddyLocator`** akzeptiert 2 Parameter, beide sind optional.

- **`numBuddies`** – bestimmt, wie viele Buddys jede Instanz wählen sollte, um ihre Daten zu sichern. Dies ist standardmäßig 1.
- **`ignoreColocatedBuddies`** – bedeutet, dass jede Instanz *versucht*, einen Buddy auf einem anderen physischen Host zu wählen. Falls dies nicht möglich ist, so fällt sie auf gemeinsam installierte Instanzen zurück. Die Standardeinstellung hier lautet **`true`**.

8.1.2.2.2. BuddyPools

Auch als *Replikationsgruppen* bekannt handelt es sich bei einem Buddy-Pool um ein optionales Konstrukt, bei dem jede Instanz in einem Cluster mit einem Buddy-Pool-Namen konfiguriert werden kann. Sie können sich dies als "exklusive Clubmitgliedschaft" vorstellen, wo bei der Auswahl von Buddys der **`BuddyLocator`**, so er denn Buddy-Pools unterstützt, Buddys wählen würde, die denselben Buddy-Pool-Namen tragen. Dies gestattet Systemadministratoren eine gewisse Flexibilität und Kontrolle darüber, wie Buddys gewählt werden. Ein Systemadministrator kann zum Beispiel Instanzen auf zwei separaten, physischen Servern, die sich auf zwei separaten, physischen Racks befinden, in denselben Buddy-Pool platzieren. Statt also eine Instanz auf einem anderen Host auf demselben Rack zu wählen, würden **`BuddyLocator`**s die Instanz im selben Buddy-Pool auf einem separaten Rack wählen, was ein gewisses Maß an Redundanz bietet.

8.1.2.2.3. Ausfallsicherheit

Falls eine Instanz abstürzt, wird angenommen, dass der sich mit dem Cache verbindende Client (direkt, oder indirekt über einen anderen Dienst wie etwa HTTP Session-Replikation) die Anfrage an jede andere zufällig gewählte Cache-Instanz im Cluster weiterleiten kann. Hier kommt das Konzept der Datengravitation ins Spiel.

Beim Konzept der Datengravitation fragt ein Cache, an den eine bestimmte Anfrage gestellt wurde, über deren Daten er jedoch selbst nicht verfügt, andere Instanzen im Cluster nach diesen Daten. Anders ausgedrückt findet hier so etwas wie eine "lazy" Übertragung der Daten statt, bei der Daten *nur dann* übertragen werden, wenn andere Knoten diese abfragen. Diese Strategie verhindert einen regelrechten Datensturm im Netzwerk, wenn große Datenmengen auf gesunden Knoten hin- und herübertragen werden, nur weil wenige oder sogar nur ein Knoten ausgefallen ist.

Werden die Daten nicht im primären Bereich eines Knotens gefunden, so würde der anfragende Knoten (optional) andere Instanzen aufgefordern, ihre für andere Caches gespeicherten Backup-Daten auf diese Daten hin zu überprüfen. Das bedeutet, dass, selbst wenn der Cache ausfällt, der Ihre Session-Daten enthält, andere Instanzen nach wie vor auf diese Daten zugreifen können, indem sie den Cluster auffordern, seine Backups nach den Daten zu durchsuchen.

Nach Auffinden der Daten werden diese an die Instanz übertragen, die sie angefragt hat, und werden dort dem Datenbaum dieser Instanz hinzugefügt. Anschließend werden sie (optional) von allen anderen Instanzen (und Backups) entfernt, so dass bei Verwendung von Session-Affinität die Affinität zu dieser neuen Cache-Instanz existiert, die diese Daten gerade *in Besitz genommen hat*.

Datengravitation wird als Interzeptor implementiert. Die folgenden Konfigurationseigenschaften (alle optional) stehen in Zusammenhang mit der Datengravitation.

- **`dataGravitationRemoveOnFind`** – zwingt alle entfernten Caches, die die Daten besitzen oder Backups enthalten, diese Daten zu löschen, wodurch der anfragende Cache zum neuen Besitzer der Daten wird. Falls auf **`false`** eingestellt, wird ein "evict" (räumen) statt eines "remove" (entfernen) gesendet, so dass im Cache Loader persistierte Zustände verbleiben. Dies ist von Nutzen, wenn ein gemeinsam verwendeter Cache Loader konfiguriert ist. Standardmäßig **`true`**.
- **`dataGravitationSearchBackupTrees`** – Teilt entfernten Instanzen mit, dass sie ihre Backups

sowie die Hauptdatenbäume durchsuchen sollen. Standardmäßig **`true`**. Wenn **`true`**, können infolgedessen auch die Backup-Knoten zusätzlich zu Dateneigentümern auf Anfragen zur Datengravitation reagieren.

- **`autoDataGravitation`** – Ob Datengravitation für jeden Cache-Fehlschlag erfolgt. Die Standardeinstellung lautet **`false`**, um unnötige Netzwerkaufrufe zu vermeiden. Die meisten Anwendungsfälle wissen, wenn die Gravitation von Daten notwendig ist und und liefern eine **`Option`**, um Datengravitation für einen einzelnen Aufruf zu aktivieren. Falls **`autoDataGravitation`** auf **`true`** gesetzt ist, so ist diese **`Option`** unnötig.

8.1.2.2.4. Konfiguration

Siehe Kapitel 12, *Konfigurationsreferenzen* für Details über die Konfiguration der Buddy-Replikation.

8.2. Invalidierung

Wird ein Cache für Invalidierung statt Replikation konfiguriert, erhalten bei jeder Änderung der Zeitdaten in einem Cache die anderen Caches im Cluster eine Nachricht, die sie darüber informiert, das ihre Daten nun veraltet sind und aus dem Speicher geräumt werden sollten. Invalidierung würde bei Verwendung mit einem gemeinsam verwendeten Cache Loader (siehe Kapitel 9, *Cache Loader*) dazu führen, dass entfernte Caches auf den gemeinsam verwendeten Cache Loader verweisen und modifizierte Daten abrufen. Dies bietet einen doppelten Vorteil: Netzwerkverkehr wird minimiert, da Invalidierungsnachrichten im Vergleich zur Replikation aktualisierter Daten sehr klein sind und andere Caches im Cluster fragen auf "lazy" Weise nach modifizierten Daten, also nur bei Bedarf.

Invalidierungsnachrichten werden nach jeder Änderung (keine Transaktionen oder Batches) verschickt oder am Ende einer Transaktion oder eines Batches nach erfolgreicher Festschreibung. Dies ist in der Regel effizienter, da Invalidierungsnachrichten für die Transaktion als Ganzes optimiert werden können statt pro Änderung.

Invalidierung kann ebenfalls synchron oder asynchron erfolgen und ganz wie bei der Replikation sperrt synchrone Invalidierung, bis alle Caches im Cluster Invalidierungsnachrichten erhalten und die veralteten Daten geräumt sind, während asynchrone Invalidierung in einem "fire-and-forget"-Modus ("senden und vergessen") stattfindet, bei dem Invalidierungsnachrichten gesendet werden, aber kein Sperren und Warten auf Antworten erfolgt.

8.3. Zustandsübertragung

Zustandsübertragung bezeichnet den Vorgang, bei dem eine JBoss Cache Instanz sich darauf vorbereitet einen Dienst bereitzustellen, indem sie den aktuellen Zustand einer anderen Cache-Instanz bezieht und diesen Zustand in ihren eigenen Zustand integriert.

8.3.1. Arten der Zustandsübertragung

Man kann die Arten von Zustandsübertragungen auf drei verschiedene Weisen unterscheiden, abhängig von der Betrachtungsweise. Erstens: Im Kontext einer bestimmten Implementierung der Zustandsübertragung, der zugrunde liegenden Vernetzung, gibt es zwei grundsätzich verschiedene Übertragungsarten: Byte-Array und Streaming-basierte Zustandsübertragung. Zweitens: Zustandsübertragung kann eine vollständige oder eine teilweise Zustandsübertragung sein, abhängig vom übertragenen Unterbaum. Die Übertragung eines kompletten Cache-Baums ist eine vollständige Übertragung, während die Übertragung eines bestimmten Unterbaums eine teilweise Zustandsübertragung ist. Drittens: Eine Zustandsübertragung kann eine "in-memory" oder "persistente" Übertragung sein, abhängig von der Verwendung des Caches.

8.3.2. Byte-Array und Streaming-basierte Zustandsübertragung

Byte-Array-basierte Übertragung war die standardmäßige und einzig verfügbare Cache-Übertragungsmethode in allen Releases bis 2.0. Die Byte-Array-basierte Übertragung lädt den gesamten, in ein Byte-Array übertragenen Zustand und sendet diesen an das empfangende Mitglied. Eine wesentliche Einschränkung dieser Herangehensweise ist die sehr umfangreiche Größe der Zustandsübertragung (>1 GB), die sehr wahrscheinlich eine "OutOfMemoryException" verursachen würde. Die Streaming-Zustandsübertragung liefert einen InputStream an einen Zustands-Reader und einen OutputStream an einen Zustands-Writer. OutputStream- und InputStream-Abstraktionen ermöglichen die Zustandsübertragung in kleineren Byte-Segmenten, haben also geringere Speicheranforderungen. Falls beispielsweise der Applikationszustand als Baum mit einer Gesamtgröße von 1 GB repräsentiert ist, muss nicht ein 1 GB großes Byte-Array übertragen werden, sondern wird durch Streaming der Zustand stattdessen als Segmente von N Bytes Größe übertragen, wobei N benutzerdefinierbar ist.

Byte-Array- und Streaming-basierte Zustandsübertragung sind vollständig API-transparent, austauschbar, und werden statisch konfiguriert durch eine standardmäßige Cache-Konfigurations-XML-Datei. Entnehmen Sie bitte der JGroups-Dokumentation, wie Sie von einer Art Übertragung zur anderen wechseln.

8.3.3. Vollständige und teilweise Zustandsübertragung

Ist entweder in-memory (speicherinterne) oder persistente Zustandsübertragung aktiviert, so wird zu verschiedenen Zeiten eine vollständige oder teilweise Zustandsübertragung durchgeführt, je nachdem, auf welche Weise der Cache verwendet wird. "Vollständige" Zustandsübertragung bedeutet die Übertragung des gesamten Baumzustandes – d.h. des Wurzelknotens und sämtlicher Knoten darunter. Bei einer "teilweisen" Zustandsübertragung wird nur ein Teil des Baums übertragen – d.h. ein Knoten an einem bestimmten Fqn und alle Knoten darunter.

Wenn die speicherinterne oder die persistente Zustandsübertragung aktiviert ist, erfolgt die Zustandsübertragung zu folgenden Zeiten:

1. Erstmalige Zustandsübertragung. Diese erfolgt, wenn der Cache erstmals gestartet wird (als Teil der Verarbeitung der **`start()`**-Methode). Es handelt sich um eine vollständige Zustandsübertragung. Der Zustand wird von der am längsten laufenden Cache-Instanz abgerufen. [1] Kommt es beim Empfang oder bei der Integration des Zustands zu einem Problem, so startet der Cache nicht.

   Die erstmalige Zustandsübertragung erfolgt, es sei denn:

   a. Die **`InactiveOnStartup`**-Eigenschaft des Caches ist **`true`**. Diese Eigenschaft wird in Verbindung mit bereichsbasiertem Marshalling verwendet; mehr dazu finden Sie weiter unten.

   b. Buddy-Replikation wird verwendet. Weiter unten finden Sie weitere Informationen zur Zustandsübertragung mit Buddy-Replikation.

2. Teilweise Zustandsübertragung nach Bereichsaktivierung. Wenn bereichsbasiertes Marshalling verwendet wird, muss die Applikation einen spezifischen Klassenlader beim Cache registrieren. Dieser Klassenlader wird zum Unmarshalling des Zustands für einen bestimmten Bereich (Unterbaum) des Caches verwendet.

   Nach der Registrierung ruft die Applikation **`cache.getRegion(fqn, true).activate()`** auf, was die teilweise Zustandsübertragung des relevanten Unterbaums iniziiert. Diese Anfrage wird zunächst an die älteste Cache-Instanz im Cluster gestellt. Falls diese Instanz jedoch mit keinem Zustand antwortet, wird die Anfrage der Reihe nach an die anderen Instanzen weitergegeben, bis entweder eine mit dem angefragten Zustand antwortet oder alle abgefragt wurden.

   Wird bereichsbasiertes Marshalling verwendet, so ist die **`InactiveOnStartup`**-Eigenschaft des

Caches in der Regel auf **`true`** eingestellt. Dies unterdrückt die erstmalige Zustandsübertragung, die fehlschlagen würde, da der übertragene Zustand nicht deserialisiert werden könnte.

3. Buddy-Replikation. Wird Buddy-Replikation verwendet, so ist die erstmalige Zustandsübertragung deaktiviert. Stattdessen wird, wenn eine Cache-Instanz sich dem Cluster anschließt, diese der Buddy einer oder mehrerer Instanzen, und eine oder mehrere Instanzen werden deren Buddy. Jedesmal, wenn eine Instanz feststellt, dass sie einen neuen Buddy als Backup hat, gibt sie ihren aktuellen Zustand an diesen neuen Buddy weiter. Dieses "Pushen" des Zustands zum neuen Buddy unterscheidet sich etwas von anderen Formen der Zustandsübertragung, die auf einem "Pull"-Vorgang basieren (d.h. der Empfänger fragt den Zustand an und empfängt diesen). Der Vorgang der Vorbereitung und Integration des Zustands ist allerdings derselbe.

   Dieser "Push" des Zustands bei der Buddy-Gruppenbildung erfolgt nur dann, wenn die **`InactiveOnStartup`**-Eigenschaft auf **`false`** eingestellt ist. Lautet die Einstellung **`true`**, so erfolgt die Zustandsübertragung unter den Buddys nur, wenn die Applikation den Bereich auf den verschiedenen Gruppenmitgliedern aktiviert.

   Auch die teilweise Zustandsübertragung nach einem Aufruf zur Bereichsaktivierung unterscheidet sich etwas im Fall der Buddy-Replikation. Statt den Teilzustand von einer Cache-Instanz abzufragen und alle Instanzen auszuprobieren, bis eine antwortet, fordert bei der Buddy-Replikation die Instanz, die einen Bereich aktiviert, den Teilzustand von jeder Instanz an, der sie als Backup dient.

8.3.4. Transiente ("in-memory" oder speicherinterne) und persistente Zustandsübertragung

Der erhaltene und integrierte Zustand kann aus zwei grundlegenden Typen bestehen:

1. Transienter (auch "in-memory" oder speicherinterner) Zustand. Bestehend aus dem tatsächlichen, speicherinternen Zustand einer anderen Cache-Instanz – die Inhalte verschiedener, speicherinterner Knoten in dem Cache, der den Zustand liefert, werden serialisiert und übertragen; der Empfänger deserialisiert die Daten, erstellt entsprechende Knoten in seinem eigenem speicherinternen Baum und füllt diese mit den übertragenen Daten.

   "In-memory" Zustandsübertragung wird durch Einstellung des **`FetchInMemoryState`**-Konfigurationsattributs des Caches auf **`true`** aktiviert.

2. "Persistenter" Zustand. Gilt nur, wenn ein nicht gemeinsam verwendeter Cache Loader verwendet wird. In dem Cache, der den Zustand liefert, wird der gespeicherte Zustand deserialisiert und übertragen; der Empfänger leitet die Daten an seinen eigenen Cache Loader weiter, der diese Daten im persistenten Speicher des Empfängers persistiert.

   "Persistente" Zustandsübertragung wird durch Einstellung des **`fetchPersistentState`**-Attributs des Cache Loaders auf **`true`** aktiviert. Sind mehrere Cache Loader in einer Kette konfiguriert, so kann diese Eigenschaft nur bei einem auf "true" eingestellt sein, da andernfalls beim Start eine Ausnahme gemeldet wird.

   Persistente Zustandsübertragung mit einem gemeinsam verwendeten Cache Loader macht keinen Sinn, da derselbe persistente Speicher, der die Daten bereitstellt, diese am Ende auch empfängt. Wird also ein Cache Loader gemeinsam verwendet, so gestattet der Cache keine persistente Zustandsübertragung, selbst wenn bei einem Cache Loader **`fetchPersistentState`** auf **`true`** eingestellt ist.

Welche Art von Zustandsübertragung am besten geeignet ist, hängt von der Verwendung des Caches ab.

1. Wird ein "Write-Through" Cache Loader verwendet, so wird der aktuelle Cache-Zustand vollständig durch den persistenten Zustand repräsentiert. Daten könnten aus dem transienten Speicher entfernt worden sein, befinden sich aber nach wie vor im persistenten Speicher. In diesem Fall wird – falls der Cache Loader nicht gemeinsam verwendet wird – persistente

Zustandsübertragung verwendet, um sicherzustellen, dass der neue Cache den korrekten Zustand besitzt. Der speicherinterne Zustand kann auch übertragen werden, wenn Sie einen "Hot" Cache möchten – einen Cache, der alle relevanten Daten im Speicher hat, wenn der Cache mit der Bereitstellung von Diensten beginnt. (Beachten Sie, dass das **<preload>**-Element im **<loaders>**-Konfigurationselement ebenfalls verwendet werden kann, um einen "warm" oder "hot" Cache bereitzustellen, ohne dass eine speicherinterne Zustandsübertragung nötig wäre. Diese Vorgehensweise reduziert geringfügig die Belastung der Cache-Instanz, die den Zustand liefert, erhöht aber die Belastung auf den persistenten Speicher auf Empfängerseite).

2. Wird ein Cache Loader mit Passivierung verwendet, so kann die vollständige Repräsentation des Zustands nur durch Kombination des speicherinternen (d.h. nicht passivierten) und persistenten (d.h. passivierten) Zustands erhalten werden. Daher ist eine speicherinternere Zustandsübertragung notwendig. Eine persistente Zustandsübertragung ist nötig, wenn der Cache Loader nicht gemeinsam verwendet wird.
3. Wird kein Cache Loader verwendet und handelt es sich beim Cache ausschließlich um einen "Write-Aside" Cache (d.h. ein Cache, der zum Cachen von Daten verwendet wird, die auch in einem persistenten Speicher, z.B. einer Datenbank vorliegen), so ist es abhängig davon, ob ein "hot" Cache erwünscht ist oder nicht, ob der speicherinterne Zustand übertragen wird oder nicht.

8.3.5. Nicht-sperrende Zustandsübertragung

Neu in JBoss Cache 3.1.0 ist die nicht-sperrende Zustandsübertragung ("Non-Blocking State Transfer" oder kurz "NBST"), die es Sendern erlaubt, Zustände zu generieren und zu streamen, ohne dabei die Verarbeitung ihrer normalen Transaktionen zu unterbrechen. Dies ist insbesondere wichtig bei umfangreichen Zuständen, bei denen das Generieren und Streamen einige Zeit in Anspruch nehmen kann, während der laufende Transaktionen auf dem Sender unter Umständen einen Timeout erreichen und fehlschlagen könnten.

Um dies zu erreichen, sollte NBST aktiviert werden (siehe Konfigurationsreferenzen), und Sie müssen MVCC als Knoten-Sperrschema einsetzen. Zusätzlich müssen Sie JGroups' STREAMING_STATE_TRANSFER-Protokoll in Ihren Cluster-Eigenschaften einsetzen.

8.3.6. Konfiguration der Zustandsübertragung

Um sicherzustellen, dass die Zustandsübertragung wie erwartet funktioniert, ist es wichtig, dass alle Knoten im Cluster mit denselben Einstellungen für persistente und transiente Zustände konfiguriert sind. Denn Byte-Array-basierte Übertragungen verlassen sich einzig auf die Konfiguration des Anfragers, während Stream-basierte Übertragungen sich sowohl auf die Konfiguration des Anfragers, als auch die des Senders verlassen und dabei davon ausgehen, dass diese identisch sind.

[1] Die am längsten laufende Cache-Instanz ist in JBoss-Begriffen immer auch der Koordinator.

Kapitel 9. Cache Loader

JBoss Cache kann einen *Cache Loader* für das Backup des speicherinternen Caches zu einem Backend-Datenspeicher verwenden. Wird JBoss Cache mit einem Cache Loader konfiguriert, stehen folgende Funktionen zur Verfügung:

- Jedesmal, wenn auf ein Cache-Element zugegriffen wird und dieses Element sich nicht im Cache befindet (z.B. durch "Eviction" oder aufgrund eines Neustarts des Servers), so lädt der Cache Loader das Element transparent in den Cache, falls er es im Backend-Speicher findet.
- Jedesmal, wenn ein Element bearbeitet, hinzugefügt oder entfernt wird, so wird diese Änderung über den Cache Loader im Backend-Speicher persistiert. Werden Transaktionen verwendet, so werden alle innerhalb einer Transaktion erstellten Änderungen persistiert. Der **`CacheLoader`** nimmt dazu teil am 2-Phasen-Festschreibungsprotokoll, das vom Transaktionsmanager ausgeführt wird, wenn auch nicht explizit.

9.1. CacheLoader-Schnittstelle und Lebenszyklus

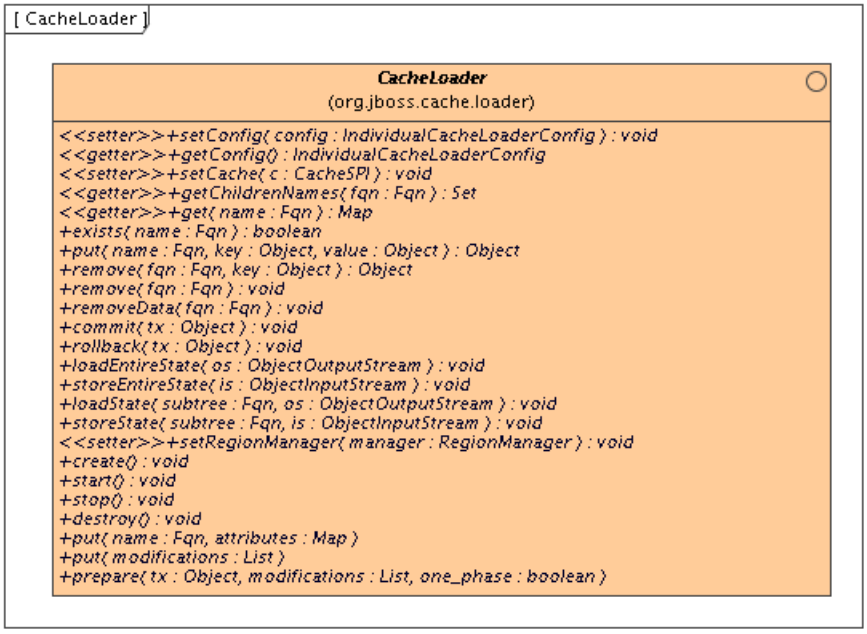

[CacheLoader]
CacheLoader
(org.jboss.cache.loader)
<<setter>>+setConfig(config : IndividualCacheLoaderConfig) : void
<<getter>>+getConfig() : IndividualCacheLoaderConfig
<<setter>>+setCache(c : CacheSPI) : void
<<getter>>+getChildrenNames(fqn : Fqn) : Set
<<getter>>+get(name : Fqn) : Map
+exists(name : Fqn) : boolean
+put(name : Fqn, key : Object, value : Object) : Object
+remove(fqn : Fqn, key : Object) : Object
+remove(fqn : Fqn) : void
+removeData(fqn : Fqn) : void
+commit(tx : Object) : void
+rollback(tx : Object) : void
+loadEntireState(os : ObjectOutputStream) : void
+storeEntireState(is : ObjectInputStream) : void
+loadState(subtree : Fqn, os : ObjectOutputStream) : void
+storeState(subtree : Fqn, is : ObjectInputStream) : void
<<setter>>+setRegionManager(manager : RegionManager) : void
+create() : void
+start() : void
+stop() : void
+destroy() : void
+put(name : Fqn, attributes : Map)
+put(modifications : List)
+prepare(tx : Object, modifications : List, one_phase : boolean)

Abbildung 9.1. Die CacheLoader-Schnittstelle

Die Interaktion zwischen JBoss Cache und einer **`CacheLoader`**-Implementierung ist wie folgt. Ist **`CacheLoaderConfiguration`** (siehe unten) nicht-Null, so wird eine Instanz eines jeden konfigurierten **`CacheLoader`** erzeugt, wenn der Cache erzeugt wird, und wird gestartet, wenn auch der Cache gestartet wird.

`CacheLoader.create()` und **`CacheLoader.start()`** werden aufgerufen, wenn der Cache gestartet wird. Entsprechend werden **`stop()`** und **`destroy()`** aufgerufen, wenn der Cache gestoppt wird.

Als Nächstes werden **`setConfig()`** und **`setCache()`** aufgerufen. Letzteres kann zum Speichern einer Referenz im Cache verwendet werden, ersteres zur Konfiguration dieser Instanz des **`CacheLoader`**. Zum Beispiel könnte hier ein Datenbank-Cache-Loader eine Verbindung mit der Datenbank herstellen.

Die **`CacheLoader`**-Schnittstelle besitzt eine Reihe von Methoden, die aufgerufen werden, wenn keine Transaktionen eingesetzt werden: **`get()`** , **`put()`** , **`remove()`** und **`removeData()`**: ein

"get"/"set"/"remove" (abrufen/einstellen/entfernen) des Werts erfolgt dadurch sofort. Diese Methoden werden als javadoc-Kommentare in der obigen Schnittstelle beschrieben.

Desweiteren werden drei Methoden mit Transaktionen verwendet: **`prepare()`** , **`commit()`** und **`rollback()`**. Die **`prepare()`**-Methode wird aufgerufen, wenn eine Transaktion festgeschrieben werden soll. Sie besitzt ein Transaktionsobjekt und eine Liste von Änderungen als Parameter. Das Transaktionsobjekt kann als ein Schlüssel in eine Hashmap von Transaktionen verwendet werden, bei der die Werte die Änderungslisten sind. Jede Änderungsliste besitzt eine Anzahl von **`Modification`**-Elementen, welche die an einem Cache für eine betreffende Transaktion vorgenommenen Änderungen repräsentiert. Wenn **`prepare()`** erfolgreich zurückkehrt, dann *muss* der Cache Loader in der Lage sein, die Transaktion erfolgreich festzuschreiben (oder zurückzusetzen).

JBoss Cache trägt dafür Sorge, dass prepare(), commit() und rollback() auf den Cache Loadern zur richtigen Zeit aufgerufen werden.

Die **`commit()`**-Methode weist den Cache Loader an, die Transaktion festzuschreiben, und die **`rollback()`**-Methode weist den Cache Loader an, die mit dieser Transaktion assoziierten Änderungen zu verwerfen.

Werfen Sie einen Blick auf die Javadocs dieser Schnittstelle für eine detaillierte Erläuterung jeder Methode und des Vertrages, den Implementierungen erfüllen müssten.

9.2. Konfiguration

Cache Loader werden folgendermaßen in der JBoss Cache XML-Datei konfiguriert. Beachten Sie, dass Sie mehrere Cache Loader in einer Kette definieren können. Der Cache schaut alle Cache Loader in deren Konfigurationsreihenfolge durch, bis er ein gültiges Datenelement findet, das nicht Null ist. Bei der Durchführung von Schreibvorgängen wird in alle Cache Loader geschrieben (es sei denn, das **`ignoreModifications`**-Element wurde für einen bestimmten Cache Loader auf **`true`** eingestellt). Einzelheiten entnehmen Sie bitte dem Konfigurationsabschnitt unten.

```
...

<!-- Cache loader config block -->
<!-- if passivation is true, only the first cache loader is used; the rest are
ignored -->
<loaders passivation="false" shared="false">
      <preload>
         <!-- Fqns to preload -->
         <node fqn="/some/stuff"/>
      </preload>
      <!-- if passivation is true, only the first cache loader is used; the rest
are ignored -->
      <loader class="org.jboss.cache.loader.JDBCCacheLoader" async="false"
fetchPersistentState="true"
               ignoreModifications="false" purgeOnStartup="false">
         <properties>
            cache.jdbc.driver=com.mysql.jdbc.Driver
            cache.jdbc.url=jdbc:mysql://localhost:3306/jbossdb
            cache.jdbc.user=root
            cache.jdbc.password=
         </properties>
      </loader>
  </loaders>
```

Das **`class`**-Element definiert die Klasse der Cache-Loader-Implementierung. (Beachten Sie dass,

aufgrund eines Fehlers im Eigenschaftseditor in JBoss Backslashes in Variablen für Windows-Dateinamen möglicherweise nicht ordnungsgemäß ersetzt werden, daher kann ein replace="false" nötig sein). Beachten Sie, dass eine Cache-Loader-Implementierung einen leeren Konstruktor haben muss.

Das **`properties`**-Element definiert eine für eine bestimmte Implementierung spezifische Konfiguration. Die Dateisystem-basierte Implementierung etwa definiert, dass das root-Verzeichnis verwendet werden soll, während eine Datenbankimplementierung die Datenbank-URL, den Namen und ein Passwort zur Herstellung einer Datenbankverbindung definieren könnte. Diese Konfiguration wird der Cache-Loader-Implementierung mittels **`CacheLoader.setConfig(Properties)`** übergeben. Beachten Sie, dass für Backspaces möglicherweise ein Maskierungszeichen nötig ist.

`preload` gestattet es uns, eine Liste von Knoten oder sogar ganzer Unterbäume zu definieren, die beim Start des Caches aufgesucht werden, um die mit diesen Knoten assoziierten Daten vorab zu laden. Der Standard ("/") lädt alle im Backend-Speicher verfügbaren Daten in den Cache, was ab einer gewissen Größe des Backend-Speichers möglicherweise keine so gute Idee ist. Zum Beispiel lädt **`/a, /product/catalogue`** die Unterbäume **`/a`** und **`/product/catalogue`** in den Cache, aber nichts anderes. Alles andere wird erst bei Bedarf geladen. Erwartet man die häufige Verwendung von Elementen in einem bestimmten Unterbaum, so ist ein Vorabladen sinnvoll.

`fetchPersistentState` legt fest, ob der persistente Zustand eines Caches abgerufen wird oder nicht, wenn er Teil eines Clusters wird. Nur ein einziger konfigurierter Cache Loader kann diese Eigenschaft auf "true" einstellen; hat mehr als ein Cache Loader diese Einstellung, so wird eine Konfigurationsausnahme gemeldet, wenn der Cache-Dienst gestartet wird.

`async` legt fest, ob Schreibvorgänge in den Cache Loader bis zum Abschluss sperren oder auf einem separaten Thread erfolgen, so dass Schreibvorgänge sofort zurückkehren. Ist dies auf "true" eingestellt, so wird eine Instanz von **`org.jboss.cache.loader.AsyncCacheLoader`** mit einer Instanz des tatsächlichen, zu verwendenden Cache Loaders konstruiert. Der **`AsyncCacheLoader`** delegiert dann alle Anfragen zum darunterliegenden Cache Loader, wobei falls nötig ein separater Thread verwendet wird. Weitere Informationen finden Sie in den Javadocs zu **`AsyncCacheLoader`**. Falls nichts festgelegt wurde, lautet der Standardwert des **`async`**-Elements *false*.

Anmerkung zur Verwendung des async-Elements

Es besteht stets die Möglichkeit von Schreib-Lese-Konflikten (sog. "Dirty Reads"), da alle Schreibvorgänge asynchron durchgeführt werden und es daher unmöglich ist zu garantieren, wann (und ob!) ein Schreibvorgang erfolgreich abgeschlossen wird. Bei der Einstellung des **`async`**-Elements auf "true" sollte dies berücksichtigt werden.

`ignoreModifications` legt fest, ob "write"-Methoden zum spezifischen Cache Loader weitergereicht werden. In manchen Situationen sollten sich transiente Applikationsdaten nur in einem dateibasierten Cache Loader auf demselben Server wie der speicherinterne Cache befinden, zum Beispiel mit einem weiteren gemeinsam verwendeten **`JDBCCacheLoader`**, der von allen Servern im Netzwerk verwendet wird. Dieses Feature gestattet es Ihnen, in den "lokalen" Datei-Cache-Loader zu schreiben, aber nicht in den gemeinsamen **`JDBCCacheLoader`**. Standardeinstellung dieser Eigenschaft ist **`false`**, so dass Schreibvorgänge an alle konfigurierten Cache Loader weitergereicht werden.

`purgeOnStatup` leert den angegebenen Cache Loader (falls **`ignoreModifications`** auf **`false`** eingestellt ist), wenn dieser startet.

`shared` zeigt an, dass dieser Cache Loader von verschiedenen Cache-Instanzen gemeinsam verwendet wird, z.B. wenn alle Instanzen in einem Cluster dieselben JDBC-Einstellungen nutzen, um mit

derselben, entfernten Datenbank zu kommunizieren. Die Einstellung auf **true** verhindert wiederholte, unnötige Schreibvorgänge derselben Daten in den Cache Loader durch unterschiedliche Cache-Instanzen. Der Standardwert lautet **false**.

9.2.1. Singleton-Store-Konfiguration

```
 <loaders passivation="false" shared="true">
    <preload>
       <node fqn="/a/b/c"/>
       <node fqn="/f/r/s"/>
    </preload>

    <!-- we can now have multiple cache loaders, which get chained -->
    <loader class="org.jboss.cache.loader.JDBCCacheLoader" async="false"
fetchPersistentState="false"
          ignoreModifications="false" purgeOnStartup="false">
       <properties>
         cache.jdbc.datasource=java:/DefaultDS
       </properties>
       <singletonStore enabled="true"
class="org.jboss.cache.loader.SingletonStoreCacheLoader">
          <properties>
             pushStateWhenCoordinator=true
             pushStateWhenCoordinatorTimeout=20000
          </properties>
       </singletonStore>
    </loader>
 </loaders>
```

Das **singletonStore**-Element erlaubt das Speichern von Änderungen auf nur einem Knoten im Cluster, dem Koordinator. Im Wesentlichen werden Daten, die auf einem Knoten eingehen, immer repliziert, damit die speicherinternen Zustände des Caches konsistent gehalten werden; der Koordinator hingegen trägt allein die Verantwortung, diese Zustände auf Festplatte zu schreiben. Diese Funktionalität kann aktiviert werden durch Setzen des **enabled**-Unterelements auf "true" in allen Knoten, doch wiegesagt nur der Koordinator des Clusters speichert Änderungen im zugrunde liegenden Cache Loader, wie im **loader**-Element definiert. Sie können nicht einen Cache Loader als **shared** definieren und gleichzeitig **singletonStore** aktiviert haben. Der Standardwert für **enabled** lautet **false**.

Optional können Sie innerhalb des **singletonStore**-Elements ein **class**-Element definieren, das die Implementierungsklasse spezifiziert, die die Singleton-Store-Funktionalität liefert. Diese Klasse muss **org.jboss.cache.loader.AbstractDelegatingCacheLoader** erweitern, und ist, falls nicht vorhanden, standardmäßig **org.jboss.cache.loader.SingletonStoreCacheLoader**.

Das **properties**-Unterelement definiert Eigenschaften, die es erlauben, das Verhalten der Klasse zu verändern, die die Singleton-Store-Funktionalität bereitstellt. Standardmäßig wurden die **pushStateWhenCoordinator**- und **pushStateWhenCoordinatorTimeout**-Eigenschaften definiert, aber es können bei Bedarf mehr hinzugefügt werden durch die benutzerdefinierte Klasse, die Singleton-Store-Funktionalität bereitstellt.

pushStateWhenCoordinator erlaubt das aktive Schreiben des speicherinternen Zustands in den Cache-Speicher, wenn ein Knoten – aufgrund einer Veränderung in der Cluster-Topologie und der Neuwahl eines Koordinators – selbst zum Koordinator wird. Dies kann sehr hilfreich sein in Situationen, in denen der Koordinator abstürzt und ein neuer Koordinator erst nach einer gewissen Verzögerung gewählt wird. Falls diese Eigenschaft auf auf **false** gesetzt wäre und der Cache aktualisiert wird, würden diese Änderungen während dieser Zeit nie persistiert werden. Ist diese Eigenschaft dagegen

auf **true** gesetzt, wird sichergestellt, dass jegliche Änderungen während dieses Vorgangs auch im Cache Loader gespeichert werden. Sie sollten diese Eigenschaft auch auf **true** setzen, wenn der Cache Loader eines jeden Knotens an einem anderen Speicherort konfiguriert ist. Der Standardwert ist **true**.

pushStateWhenCoordinatorTimeout ist nur relevant, wenn **pushStateWhenCoordinator true** ist. Ist dies der Fall, stellt es die maximale Anzahl von Millisekunden ein, die der Vorgang, den speicherinternen Zustand in den zugrunde liegenden Cache Loader zu schreiben, dauern sollte. Eine **PushStateException** wird gemeldet, wenn diese Grenze überschritten wird. Der Standardwert ist 20000.

Anmerkung zur Verwendung des singletonStore-Elements

Das Einrichten eines Cache Loaders als Singleton und das Verwenden von Cache-Passivierung (mittels Eviction) kann zu unerwünschten Ergebnissen führen. Wenn ein Knoten aufgrund einer Eviction passiviert werden soll, während der Cluster gerade dabei ist, einen neuen Koordinator zu wählen, gehen die Daten verloren. Der Grund hierfür ist, dass zu diesem Zeitpunkt kein Koordinator aktiv ist und daher keiner der Knoten im Cluster den passivierten Knoten speichern wird. Ein neuer Koordinator wird im Cluster gewählt, wenn entweder der bisherige Koordinator den Cluster verlässt, er abstürzt oder nicht mehr antwortet.

9.3. Standardmäßig enthaltene Implementierungen

Die derzeit mit JBoss Cache mitgelieferten Implementierungen sind:

9.3.1. Dateibasierte Cache Loader

JBoss Cache wird mit mehreren Cache Loadern geliefert, die das Dateisystem als Datenspeicher nutzen. Sie alle erfordern, dass das **<loader><properties>**-Konfigurationselement eine **location**-Eigenschaft enthält, die auf ein Verzeichnis verweist, das als persistenter Speicher verwendet werden soll (z.B. **location=/tmp/myDataStore**). Dies wird hauptsächlich zu Testzwecken verwendet und ist nicht für den Einsatz in Produktionsumgebungen empfohlen.

- **FileCacheLoader**, eine einfache dateisystembasierte Implementierung. Standardmäßig prüft dieser Cache Loader hinsichtlich möglicher Probleme mit der Portabilität von Zeichen in Speicher- oder Baumknotennamen, z.B. wegen ungültiger Zeichen, und gibt Warnmeldungen aus. Diese Überprüfung kann deaktiviert werden, indem Sie die **check.character.portability**-Eigenschaft zum **<properties>**-Element hinzufügen und diese auf **false** einstellen (z.B. **check.character.portability=false**).

  Der FileCacheLoader hat einige wesentliche Nachteile, die ihn für den Einsatz in einer Produktionsumgebung nur eingeschränkt brauchbar machen. Möchten Sie ihn dennoch in einer Produktionsumgebung einsetzen, sollten Sie sich dieser Einschränkungen unbedingt bewusst sein und entsprechend Vorsicht walten lassen.

  - Aufgrund der Art und Weise, wie der FileCacheLoader eine Baumstruktur auf der Festplatte repräsentiert (Verzeichnisse und Dateien), ist Traversal für tief verschachtelte Bäume ineffizient.
  - Der Einsatz auf freigegebenen Dateisystemen wie NFS, Windows Shares, etc. sollte vermieden werden, da diese keine ordnungsgemäßen Dateisperren implementieren und beschädigte Daten die Folge sein könnten.
  - Die Verwendung mit der Isolationsebene NONE kann zu fehlerhaften Schreibvorgängen führen, da mehrere Threads versuchen, in dieselbe Datei zu schreiben.

- Dateisysteme sind naturgemäß nicht transaktional, wenn Sie also versuchen, Ihren Cache in einem transaktionalen Kontext zu verwenden, können Fehler beim Schreiben in die Datei (die während der Festschreibungsphase auftreten) nicht wiederhergestellt werden.

Als Faustregel empfehlen wir Ihnen, den FileCacheLoader nicht in einer transaktionalen, hoch nebenläufigen oder sehr beanspruchten Umgebung einzusetzen, sondern ihn nur zu Testzwecken zu verwenden.

- **`BdbjeCacheLoader`**, eine Cache-Loader-Implementierung basierend auf Oracle/Sleepycat's BerkeleyDB Java Edition.
- **`JdbmCacheLoader`**, eine Cache-Loader-Implementierung basierend auf der JDBM Engine, einer schnellen und kostenlosen Alternative zu BerkeleyDB.

Beachten Sie, dass die BerkeleyDB-Implementierung wesentlich effizienter als die dateisystembasierte Implementierung ist und transaktionale Garantien bietet, jedoch eine kommerzielle Lizenz benötigt, falls sie mit einer Applikation vertrieben wird (siehe http://www.oracle.com/database/berkeley-db/index.html für Details).

9.3.2. Cache Loader, die an andere Caches delegieren

- **`LocalDelegatingCacheLoader`**, ermöglicht das Laden aus bzw. Speichern in einem anderen lokalen Cache (in derselben VM).
- **`ClusteredCacheLoader`**, gestattet die Abfrage von speicherinternen Daten anderer Caches im selben Cluster über dieselben Clustering-Protokolle, die auch zur Replikation von Daten verwendet werden. Schreibvorgänge werden jedoch **nicht** gespeichert, da sich die Replikation um eventuell nötige Aktualisierungen kümmern würde. Sie müssen eine Eigenschaft namens **`timeout`** festlegen, einen Long-Wert, der dem Cache Loader mitteilt, wieviele Millisekunden auf Antwort vom Cluster gewartet werden soll, ehe von einem Nullwert ausgegangen wird. Zum Beispiel würde **`timeout = 3000`** einen Timeout-Wert von 3 Sekunden festlegen.

9.3.3. JDBCCacheLoader

JBossCache wird mit einer JDBC-basierten Cache-Loader-Implementierung vertrieben, die die Zustände der Knoten in einer relationalen Datenbank speichert/lädt. Die implementierende Klasse ist **`org.jboss.cache.loader.JDBCCacheLoader`**.

Die aktuelle Implementierung verwendet nur eine Tabelle. Jede Reihe in der Tabelle steht für einen Knoten und umfasst drei Spalten:

- Spalte für **`Fqn`** (gleichzeitig eine Primärschlüsselspalte)
- Spalte für Knoteninhalte (Attribut-/Wertpaare)
- Spalte für übergeordneten **`Fqn`**

`Fqn`s werden als Zeichenketten gespeichert. Knoteninhalte werden als BLOB gespeichert.

Warnung

JBoss Cache erlegt den in Fqn verwendeten Objekttypen keine Einschränkungen auf, aber diese Cache-Loader-Implementierung erfordert, dass Fqn nur Objekte des Typs **`java.lang.String`** enthält. Eine weitere Einschränkung für **`Fqn`** ist dessen Länge. Da es sich bei **`Fqn`** um einen Primärschlüssel handelt, lautet sein standardmäßiger Spaltentyp **`VARCHAR`**, der Textwerte bis zu einer bestimmten, durch die Datenbank festgelegten Höchstlänge speichern kann.

Werfen Sie einen Blick auf diese Wiki-Seite für Konfigurationstipps mit spezifischen Datenbanksystemen.

9.3.3.1. JDBCCacheLoader-Konfiguration

9.3.3.1.1. Tabellenkonfiguration

Tabellen- und Spaltennamen sowie Spaltentypen sind mit den folgenden Eigenschaften konfigurierbar.

- *cache.jdbc.table.name* – der Name der Tabelle. Dem Namen kann ein Schemaname vorangestellt werden für die angegebene Tabelle: **`{schema_name}.{table_name}`**. Der Standardwert lautet "jbosscache".
- *cache.jdbc.table.primarykey* – der Name des Primärschlüssels für die Tabelle. Der Standardwert lautet "jbosscache_pk".
- *cache.jdbc.table.create* – kann "true" oder "false" sein. Gibt an, ob die Tabelle während des Starts erstellt werden soll. Falls "true" wird die Tabelle erstellt, wenn sie nicht bereits existiert. Die Standardeinstellung ist "true".
- *cache.jdbc.table.drop* – kann "true" oder "false" sein. Gibt an, ob die Tabelle während des Beendens verworfen wird. Die Standardeinstellung lautet "true".
- *cache.jdbc.fqn.column* – FQN-Spaltenname. Der Standardwert lautet "fqn".
- *cache.jdbc.fqn.type* – FQN-Spaltentyp. Der Standardwert ist "varchar(255)".
- *cache.jdbc.node.column* – Spaltenname der Knoteninhalte. Der Standardwert lautet "node".
- *cache.jdbc.node.type* – Spaltentyp der Knoteninhalte. Der Standardwert lautet "blob". Dieser Typ muss einen gültigen Binärdatentyp für die verwendete Datenbank festlegen.

9.3.3.1.2. Datenquelle

Wenn Sie JBossCache in einer verwalteten Umgebung verwenden (z.B. einem Applikationsserver), so können Sie den JNDI-Namen der Datenquelle festlegen, die Sie verwenden wollen.

- *cache.jdbc.datasource* – JNDI-Name der Datenquelle. Der Standardwert lautet **`java:/DefaultDS`**.

9.3.3.1.3. JDBC-Treiber

Wenn Sie *keine* Datenquelle verwenden, stehen Ihnen die folgenden Eigenschaften zur Konfiguration des Datenbankzugriffs mittels JDBC-Treiber zur Verfügung.

- *cache.jdbc.driver* – vollqualifizierter JDBC-Treibername.
- *cache.jdbc.url* – URL zur Verbindung mit der Datenbank.
- *cache.jdbc.user* – Benutzername zur Verbindung mit der Datenbank.
- *cache.jdbc.password* – Passwort zur Verbindung mit der Datenbank.

9.3.3.1.4. c3p0-Verbindungs-Pooling

JBoss Cache implementiert JDBC-Verbindungs-Pooling, wenn außerhalb eines Applikationsservers eigenständig ausgeführt, unter Verwendung der c3p0:JDBC-Datenquellen/Ressourcen-Pool-Bibliothek. Um es zu aktivieren, bearbeiten Sie einfach die folgende Eigenschaft:

- *cache.jdbc.connection.factory* – Klassenname der Verbindungs-Factory. Falls nicht eingestellt, wird die standardmäßige, nicht-gepoolte Implementierung verwendet. Um c3p0-Pooling zu aktivieren, stellen Sie einfach die Verbindungs-Factory-Klasse auf c3p0. Siehe Beispiel unten.

Sie können c3p0-Parameter auch in demselben Cache-Loader-Eigenschaftsabschnitt einstellen, vergessen Sie dabei aber nicht, den Eigenschaftsnamen mit "c3p0." zu beginnen. Eine detaillierte Liste aller verfügbaren Eigenschaften finden Sie in der c3p0-Dokumentation für die c3p0-Bibliotheksversion,

die unter c3p0:JDBC DataSources/Resource Pools vertrieben wird. Für das einfache und schnelle Ausprobieren verschiedener Pooling-Parameter kann jede dieser Eigenschaften auch mittels der Systemeigenschaft eingestellt werden, welche jegliche Werte außer Kraft setzt, die diese Eigenschaft unter Umständen in der JBoss Cache XML-Konfigurationsdatei hat, z.B.: **`-Dc3p0.maxPoolSize=20`**. Ist eine c3p0-Eigenschaft weder in der Konfigurationsdatei noch als Systemeigenschaft konfiguriert, greifen die Standardwerte, wie in der c3p0-Dokumentation ausgewiesen.

9.3.3.1.5. Konfigurationsbeispiel

Nachfolgend sehen Sie ein Beispiel eines JDBCCacheLoader unter Verwendung von Oracle als Datenbank. Das CacheLoaderConfiguration XML-Element enthält einen beliebigen Satz von Eigenschaften, welche die Konfiguration im Hinblick auf die Datenbank definieren.

```
<loaders passivation="false" shared="false">
      <preload>
         <node fqn="/some/stuff"/>
      </preload>
      <!-- if passivation is true, only the first cache loader is used; the rest
are ignored -->
      <loader class="org.jboss.cache.loader.JDBCCacheLoader" async="false"
fetchPersistentState="true"
              ignoreModifications="false" purgeOnStartup="false">
         <properties>
            cache.jdbc.table.name=jbosscache
            cache.jdbc.table.create=true
            cache.jdbc.table.drop=true
            cache.jdbc.table.primarykey=jbosscache_pk
            cache.jdbc.fqn.column=fqn
            cache.jdbc.fqn.type=VARCHAR(255)
            cache.jdbc.node.column=node
            cache.jdbc.node.type=BLOB
            cache.jdbc.parent.column=parent
            cache.jdbc.driver=oracle.jdbc.OracleDriver
            cache.jdbc.url=jdbc:oracle:thin:@localhost:1521:JBOSSDB
            cache.jdbc.user=SCOTT
            cache.jdbc.password=TIGER
          </properties>
      </loader>
  </loaders>
```

Als Alternative zur Konfiguration der gesamten JDBC-Verbindung kann der Name einer bestehenden Datenquelle eingegeben werden:

```
 <loaders passivation="false" shared="false">
      <preload>
         <node fqn="/some/stuff"/>
      </preload>
      <!-- if passivation is true, only the first cache loader is used; the rest
are ignored -->
      <loader class="org.jboss.cache.loader.JDBCCacheLoader" async="false"
fetchPersistentState="true"
              ignoreModifications="false" purgeOnStartup="false">
         <properties>
            cache.jdbc.datasource=java:/DefaultDS
         </properties>
      </loader>
  </loaders>
```

Konfigurationsbeispiel für einen Cache Loader, der c3p0-JDBC-Verbindungs-Pooling nutzt:

```
 <loaders passivation="false" shared="false">
      <preload>
         <node fqn="/some/stuff"/>
      </preload>
      <!-- if passivation is true, only the first cache loader is used; the rest
are ignored -->
      <loader class="org.jboss.cache.loader.JDBCCacheLoader" async="false"
fetchPersistentState="true"
               ignoreModifications="false" purgeOnStartup="false">
         <properties>
            cache.jdbc.table.name=jbosscache
            cache.jdbc.table.create=true
            cache.jdbc.table.drop=true
            cache.jdbc.table.primarykey=jbosscache_pk
            cache.jdbc.fqn.column=fqn
            cache.jdbc.fqn.type=VARCHAR(255)
            cache.jdbc.node.column=node
            cache.jdbc.node.type=BLOB
            cache.jdbc.parent.column=parent
            cache.jdbc.driver=oracle.jdbc.OracleDriver
            cache.jdbc.url=jdbc:oracle:thin:@localhost:1521:JBOSSDB
            cache.jdbc.user=SCOTT
            cache.jdbc.password=TIGER

cache.jdbc.connection.factory=org.jboss.cache.loader.C3p0ConnectionFactory
            c3p0.maxPoolSize=20
            c3p0.checkoutTimeout=5000
         </properties>
      </loader>
   </loaders>
```

9.3.4. S3CacheLoader

Der **`S3CacheLoader`** nutzt die Amazon S3 (Simple Storage Solution) zum Speichern von Cache-Daten. Da es sich bei Amazon S3 um entfernten Netzwerkspeicher handelt und als solches eine ziemlich hohe Latenz hat, ist er am besten geeignet für Caches, die große Daten wie z.B. Media oder Dateien speichern. Dennoch sollten Sie diesen Cache Loader statt des JDBC oder der dateibasierten Cache Loader in Betracht ziehen, wenn Sie entfernt verwalteten, hochverfügbaren Speicher benötigen. Oder verwenden Sie ihn für Applikationen, die in Amazon's EC2 (Elastic Compute Cloud) laufen.

Wenn Sie Amazon S3 als Speicher verwenden möchten, sollten Sie erwägen, ihn zusammen mit JBoss Cache einzusetzen. JBoss Cache bietet selbst speicherinternes Caching Ihrer Daten, so dass die Anzahl der Remote-Aufrufe reduziert wird, um damit auch die Latenz und den Aufwand des Datenabrufs von Amazon S3 zu verringern. Mit Cache-Replikation können Sie darüberhinaus Daten von Ihrem lokalen Cluster laden, ohne jedesmal von Remote aus darauf zugreifen zu müssen.

Beachten Sie, dass Amazon S3 keine Transaktionen unterstützt. Wenn in Ihrer Applikationen Transaktionen eingesetzt werden, besteht bei der Verwendung dieses Cache Loaders das Risiko inkonsistenter Zustände. Schreibvorgänge sind jedoch atomisch, das heißt, falls ein Schreibvorgang fehlschlägt, wird der gesamte Schreibvorgang rückgängig gemacht und die Daten werden somit nie verfälscht.

Daten werden in Schlüsseln basierend auf dem Fqn des Knotens gespeichert, und Knotendaten werden als eine java.util.Map unter Verwendung der **`CacheSPI.getMarshaller()`**-Instanz serialisiert. In den Javadocs finden Sie weitere Informationen über die Strukturierung und Speicherung von Daten. Daten

werden unter Verwendung der Java-Serialisierung gespeichert. Beachten Sie, dass dadurch die Daten nicht einfach über HTTP für nicht-JBoss-Cache-Clients zur Verfügung stehen. Wir würden uns über Ihre Rückmeldung und Hilfe freuen, um diesen Cache Loader dahingehend zu erweitern.

Bei diesem Cache Loader sind einzelne Schlüsseloperationen wie **`Node.remove(Object)`** und **`Node.put(Object, Object)`** die langsamsten, da Daten in einer einzelnen Map-Instanz gespeichert werden. Verwenden Sie Massenoperationen wie **`Node.replaceAll(Map)`** und **`Node.clearData()`** für mehr Effizienz. Probieren Sie auch die **`cache.s3.optimize`**-Option aus.

9.3.4.1. Amazon S3 Bibliothek

Der S3 Cache Loader wird mit der Standarddistribution vertrieben, erfordert jedoch eine Bibliothek, um zur Laufzeit auf den Dienst zuzugreifen. Diese Laufzeitbibliothek steht in einem Sourceforge Maven Repository zur Verfügung. Fügen Sie folgende Zeilen in Ihre pom.xml-Datei ein:

```
<repository>
   <id>e-xml.sourceforge.net</id>
   <url>http://e-xml.sourceforge.net/maven2/repository</url>
</repository>
  ...
<dependency>
   <groupId>net.noderunner</groupId>
   <artifactId>amazon-s3</artifactId>
   <version>1.0.0.0</version>
   <scope>runtime</scope>
</dependency>
```

Auch wenn Sie Maven nicht verwenden, können Sie dennoch die Amazon S3 Bibliothek herunterladen, entweder indem Sie das Repository durchsuchen oder über diese URL.

9.3.4.2. Konfiguration

Sie müssen mindestens Ihren Amazon S3 Zugriffsschlüssel und geheimen Zugriffsschlüssel konfigurieren. Die folgenden Konfigurationsschlüssel sind in der Reihenfolge ihres Nutzens aufgeführt.

- **`cache.s3.accessKeyId`** – Amazon S3 Zugriffsschlüssel, erhältlich in Ihrem Account-Profil.
- **`cache.s3.secretAccessKey`** – Amazon S3 geheimer Zugriffsschlüssel, erhältlich in Ihrem Account-Profil. Da es sich hierbei um ein Passwort handelt, gehen Sie vorsichtig damit um; geben Sie diesen geheimen Schlüssel nicht weiter und binden Sie ihn nicht in in erzeugte Software ein.
- **`cache.s3.secure`** – Der Standard lautet **`false`**: Daten werden unverschlüsselt über das öffentliche Internet gesendet. Setzen Sie es auf **`true`**, um HTTPS zu verwenden. Beachten Sie, dass unverschlüsselte Uploads und Downloads die CPU weniger beanspruchen.
- **`cache.s3.bucket`** – Name des "Buckets" zur Speicherung von Daten. Verwenden Sie für unterschiedliche Caches, die denselben Zugriffsschlüssel nutzen, unterschiedliche Bucket-Namen. In der S3-Dokumentation finden Sie die Definition eines "Buckets". Der Standardwert lautet **`jboss-cache`**.
- **`cache.s3.callingFormat`** – Entweder **`PATH`**, **`SUBDOMAIN`** oder **`VANITY`**. In der S3-Dokumentation finden Sie Informationen über den Aufruf von Domains. Der Standard lautet **`SUBDOMAIN`**.
- **`cache.s3.optimize`** – Der Standard lautet **`false`**. Falls "true", ersetzen **`put(Map)`**-Operationen die an einem Fqn gespeicherten Daten, statt zu versuchen, diese abzurufen und zusammenzuführen. (Diese Option ist derzeit noch experimentell.)
- **`cache.s3.parentCache`** – Der Standard lautet **`true`**. Setzen Sie diesen Wert auf **`false`**, wenn

Sie mehrere Caches nutzen, die denselben S3-Bucket verwenden, die übergeordnete Knoten der Knoten entfernen, die in anderen Caches erzeugt wurden. (Dies ist kein üblicher Anwendungsfall.)

JBoss Cache speichert Knoten in einem Baumformat und erzeugt dabei automatisch übergeordnete Knoten, wo nötig. Der S3 Cache Loader muss diese übergeordneten Knoten ebenfalls erzeugen, damit Operationen wie **`getChildrenNames`** ordnungsgemäß funktionieren können. Das Überprüfen, ob für jede **`put`**-Operation alle übergeordneten Knoten existieren, ist ziemlich ressourcenintensiv, weshalb der Cache Loader standardmäßig das Vorhandensein dieser übergeordneten Knoten cacht.

- **`cache.s3.location`** – Dies wählt einen primären Speicherort für Ihre Daten, um die Latenz beim Abrufen und Laden zu verringern. Setzen Sie dies auf **`EU`**, um Daten in Europa zu speichern. Der Standard ist **`null`**, um Daten in den Vereinigten Staaten zu speichern.

9.3.5. TcpDelegatingCacheLoader

Dieser Cache Loader gestattet das Delegieren von Lade- und Speichervorgängen an eine andere Instanz von JBossCache, die sich (a) im selben Adressbereich, (b) in einem anderen Prozess im selben Host oder (c) in einem anderen Prozess in einem anderen Host befindet.

Ein TcpDelegatingCacheLoader kommuniziert mit einem entfernten **`org.jboss.cache.loader.tcp.TcpCacheServer`**, der ein von der Befehlszeile aus gestarteter, eigenständiger Prozess sein kann oder aber als ein MBean in JBoss AS eingebettet sein kann. Der **`TcpCacheServer`** besitzt eine Referenz zu einer anderen JBoss Cache-Instanz, die er selbst erstellen kann oder die ihm zur Verfügung gestellt wird (z.B. von JBoss mittels Abhängigkeitsinjektion).

Ab JBoss Cache 2.1.0 handhabt der **`TcpDelegatingCacheLoader`** transparent den erneuten Verbindungsaufbau, falls die Verbindung zum TcpCacheServer abbricht.

Der TcpDelegatingCacheLoader wird mit dem Host und dem Port des entfernten TcpCacheServer konfiguriert, was er zur Kommunikation mit demselben verwendet. Zusätzlich werden zwei neue, optionale Parameter verwendet, um die transparente Neuverbindung mit dem TcpCacheServer zu steuern. Die **`timeout`**-Eigenschaft (standardmäßig 5000) legt die Zeitspanne fest, während der der Cache Loader weiterhin versuchen muss, mit dem TcpCacheServer zu verbinden, bevor er aufgeben und eine Ausnahme ausgeben soll. Die **`reconnectWaitTime`**-Eigenschaft (standardmäßig 500) legt fest, wie lange der Cache Loader warten soll, bevor er versucht, sich wieder zu verbinden, wenn er einen Kommunikationsfehler entdeckt. Die letzten zwei Parameter können dazu genutzt werden, um eine Ebene der Fehlertoleranz zum Cache Loader hinzuzufügen, um mit TcpCacheServer-Neustarts umzugehen.

Die Konfiguration sieht folgendermaßen aus:

```
 <loaders passivation="false" shared="false">
      <preload>
         <node fqn="/"/>
      </preload>
      <!-- if passivation is true, only the first cache loader is used; the rest
are ignored -->
      <loader class="org.jboss.cache.loader.TcpDelegatingCacheLoader">
         <properties>
            host=myRemoteServer
            port=7500
            timeout=10000
            reconnectWaitTime=250
         </properties>
      </loader>
   </loaders>
```

Das bedeutet, dass diese Instanz von JBoss Cache alle Lade-und Speicheranfragen an den entfernten TcpCacheServer delegiert, der auf **`myRemoteServer:7500`** läuft.

Ein typischer Anwendungsfall wären mehrere, replizierte Instanzen von JBoss Cache in demselben Cluster, die alle zu derselben TcpCacheServer-Instanz delegieren. Der TcpCacheServer könnte via JDBCCacheLoader selbst zu einer Datenbank delegieren, aber der Punkt hier ist, dass – wenn fünf Knoten alle auf denselben Datensatz zugreifen – diese die Daten vom TcpCacheServer laden, der eine SQL-Anweisung pro entladenem Datensatz ausführen muss. Gingen die Knoten direkt in die Datenbank, so würde dieselbe SQL mehrere Male ausgeführt. Daher fungiert TcpCacheServer als natürlicher Cache vor der Datenbank (in der Annahme, dass ein Netzwerk-Rundtrip hin und zurück schneller ist als ein Datenbankzugriff, der in der Regel ebenfalls einen Netzwerk-Rundtrip hin und zurück beinhaltet).

Um zu vermeiden, dass ein Fehler an dieser Stelle das gesamte System zum Scheitern bringen könnte (sog. "single point of failure"), könnten wir mehrere Cache Loader konfigurieren. Der erste Cache Loader ist ein ClusteredCacheLoader, der zweite ein TcpDelegatingCacheLoader und der dritte und letzte ein JDBCacheLoader, was unsere Zugriffskosten auf einen Cache in aufsteigender Reihenfolge definiert.

9.3.6. Umwandlung von Cache Loadern

Die Art und Weise, wie gecachte Daten in die auf **`FileCacheLoader`** und **`JDBCCacheLoader`** basierenden Cache-Speicher geschrieben werden, hat sich in JBoss Cache 2.0 insofern geändert, als diese Cache Loader nun Daten schreiben und lesen unter Verwendung desselben Marhalling-Frameworks, das auch zur Replikation von Daten über das Netzwerk hinweg eingesetzt wird. Diese Änderung ist hinsichtlich der Replikation trivial, denn die anderen Knoten müssen einfach nur dieses Format verstehen. Allerdings bringt die Änderung des Datenformats in Cache-Speichern ein anderes Problem mit sich: Wie können Benutzer, die ihre Daten im JBoss Cache 1.x.x Format gespeichert haben, ihre Speicher nun zum JBoss Cache 2.0 Format migrieren?

Aus diesem Grund wird JBoss Cache 2.0 mit zwei Cache-Loader-Implementierungen namens **`org.jboss.cache.loader.TransformingFileCacheLoader`** und **`org.jboss.cache.loader.TransformingJDBCCacheLoader`** ausgeliefert, die sich innerhalb der optionalen jbosscache-cacheloader-migration.jar-Datei befinden. Dabei handelt es sich um besondere Cache Loader, die Daten aus dem Cache-Speicher im JBoss Cache 1.x.x Format lesen und Daten in Cache-Speicher im JBoss Cache 2.0 Format schreiben.

Die Idee dabei ist, dass Benutzer ihre vorhandene(n) Cache-Konfigurationsdatei(en) vorübergehend ändern, um diese Cache Loader zu verwenden, und dass sie eine kleine Java-Applikation erstellen, die eine Instanz dieses Caches erzeugt, rekursiv den gesamten Cache ausliest und die Daten wieder in den Cache zurückschreibt. Sobald die Daten umgewandelt sind, können die Benutzer wieder zu ihren ursprünglichen Cache-Konfigurationsdateien zurückkehren. Um Benutzern bei dieser Aufgabe zu helfen, wurde ein Beispiel zur Cache-Loader-Migration konstruiert, das sich unter dem **`examples/cacheloader-migration`**-Verzeichnis innerhalb der JBoss Cache-Distribution befindet. Dieses Beispiel namens **`examples.TransformStore`** ist unabhängig von den tatsächlich im Cache gespeicherten Daten, da es in den Cache zurückschreibt, was es zuvor rekursiv ausgelesen hat. Allen Benutzern, die ihre Daten portieren möchten, wird dringend empfohlen, vorher dieses Beispiel durchzuführen und es als Ausgangspunkt für ihre eigene Applikation zu verwenden. Im Beispiel finden Sie auch eine **`readme.txt`**-Datei mit detaillierten Informationen über das Beispiel selbst.

9.4. Cache-Passivierung

Ein CacheLoader kann dazu verwendet werden, die Knotenpassivierung und -aktivierung bei Eviction in einem Cache zu erzwingen.

Cache-Passivierung heißt der Vorgang, bei dem ein Objekt aus dem speicherinternen Cache entfernt

wird und dies beim Räumungsvorgang ("Eviction") stattdessen in einen sekundären Datenspeicher (z.B. Dateisystem, Datenbank) zu schreiben. *Cache-Aktivierung* heißt der Vorgang, bei dem ein Objekt aus dem Datenspeicher wieder im speicherinternen Cache hergestellt wird, wenn dieses Objekt gebraucht wird. In beiden Fällen wird der konfigurierte Cache Loader für das Lesen aus dem Datenspeicher und das Schreiben in den Datenspeicher verwendet.

Wenn eine Eviction-Richtlinie tatsächlich einen Knoten aus dem Cache räumt und Passivierung aktiviert ist, wird eine Benachrichtigung über die Passivierung des Knotens an die Cache-Listener herausgegeben und der Knoten und dessen untergeordneten Elemente werden im Cache-Loader-Speicher gespeichert. Wenn ein Benutzer versucht, einen zuvor aus dem Cache geräumten Knoten abzurufen, wird der Knoten aus dem Cache-Loader-Speicher in den speicherinternen Cache geladen (sog. "lazy load", Laden bei Bedarf). Nachdem der Knoten und dessen untergeordneten Elemente geladen wurden, werden diese aus dem Cache Loader entfernt und eine Benachrichtigung über die Aktivierung des Knotens wird an die Cache-Listener herausgegeben.

Um die Cache-Passivierung/-Aktivierung zu aktivieren, können Sie **`passivation`** auf "true" setzen. Die Standardeinstellung lautet **`false`**. Wird Passivierung verwendet, so wird nur der erste konfigurierte Cache Loader benutzt. Alle anderen werden ignoriert.

9.4.1. Cache-Loader-Verhalten mit aktivierter bzw. deaktivierter Passivierung

Ist Passivierung deaktiviert, wird, wann immer ein Element verändert, hinzugefügt oder entfernt wird, diese Veränderung im Backend-Speicher via dem Cache Loader persistiert. Es gibt keinen direkten Zusammenhang zwischen Eviction und Cache Loading. Wenn Sie Eviction nicht einsetzen, ist der Inhalt des persistenten Speichers im Grunde nur eine Kopie der speicherinternen Inhalte. Wenn Sie Eviction einsetzen, ist der Inhalt des persistenten Speichers eine Obermenge der speicherinternen Inhalte (d.h. es enthält noch Knoten, die speicherintern bereits verworfen wurden).

Ist Passivierung aktiviert, gibt es einen direkten Zusammenhang zwischen Eviction und Cache Loading. Schreibvorgänge in den persistenten Speicher via dem Cache Loader erfolgen nur als Teil des Eviction-Vorgangs. Daten werden aus dem persistenten Speicher gelöscht, wenn die Applikation diese zurück in den Arbeitsspeicher einliest. In diesem Fall handelt es sich bei den Inhalten des Arbeitsspeichers und den Inhalten des persistenten Speichers jeweils um eine Teilmenge der Gesamtinformationsmenge, wobei diese Teilmengen keine gemeinsame Schnittmenge besitzen.

Sehen Sie nachfolgend ein einfaches Beispiel, das veranschaulicht, welche Zustände sich im Verlaufe eines 6-Schritte-Vorgangs jeweils im Arbeitsspeicher bzw. im persistenten Speicher befinden:

1. Einfügen von /A
2. Einfügen von /B
3. Eviction-Thread wird ausgeführt, räumt /A
4. Lesen von /A
5. Eviction-Thread wird ausgeführt, räumt /B
6. Entfernen von /B

Bei deaktivierter Passivierung:

```
1) Memory: /A Disk: /A
2) Memory: /A, /B Disk: /A, /B
3) Memory: /B Disk: /A, /B
4) Memory: /A, /B Disk: /A, /B
5) Memory: /A Disk: /A, /B
6) Memory: /A Disk: /A
```

Bei aktivierter Passivierung:

```
1) Memory: /A Disk:
2) Memory: /A, /B Disk:
3) Memory: /B Disk: /A
4) Memory: /A, /B Disk:
5) Memory: /A Disk: /B
6) Memory: /A Disk:
```

9.5. Strategien

Dieser Abschnitt behandelt verschiedene Muster zur Kombination unterschiedlicher Cache-Loader-Typen und Konfigurationsoptionen, um bestimmte Ergebnisse zu erzielen.

9.5.1. Lokaler Cache mit Speicher

Dies ist der einfachste Fall. Wir haben eine JBoss Cache-Instanz, deren Cache-Modus **`LOCAL`** ist, weshalb keine Replikation stattfindet. Der Cache Loader lädt einfach nicht vorhandene Elemente aus dem Speicher und speichert Änderungen wieder im Speicher. Wird der Cache gestartet, können, abhängig vom **`preload`**-Element, bestimmte Daten vorab geladen werden, so dass der Cache einen "Warmstart" hat.

9.5.2. Replizierte Caches, bei dem alle Knoten denselben Speicher verwenden

Die folgende Abbildung zeigt zwei JBoss Cache-Instanzen, die sich denselben Backend-Speicher teilen:

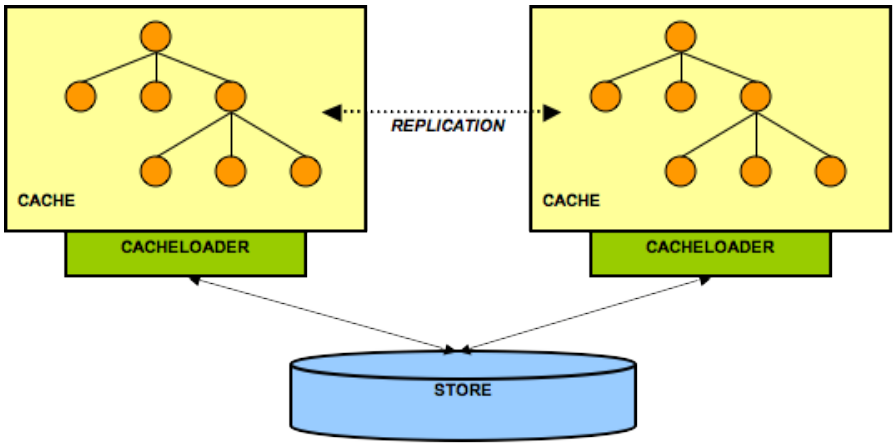


Abbildung 9.2. Zwei Nodes teilen sich einen Backend-Speicher

Beide Knoten haben einen Cache Loader, der auf einen gemeinsam verwendeten Backend-Speicher zugreift. Dies könnte beispielsweise ein freigegebenes Dateisystem sein (mittels FileCacheLoader), oder eine gemeinsam verwendete Datenbank. Da beide Knoten auf denselben Speicher zugreifen, ist eine Zustandsübertragung beim Start nicht unbedingt notwendig.

Anmerkung

Natürlich ist es möglich, die Zustandsübertragung zu aktivieren, wenn nach dem Start ein "Warm-" oder "Heißstart" des Caches gewünscht wird.

Das **`FetchInMemoryState`**-Attribut kann auf "false" eingestellt werden, was zu einem "Kaltstart" des Caches führt, der durch Zugriffe und erstmaliges Laden von Elementen zunehmend "warm" wird. Dies würde bedeuten, dass einzelne Caches innerhalb eines Clusters über einen unterschiedlichen

speicherinternen Zustand verfügen könnten (im Wesentlichen in Abhängigkeit von den verwendeten Preloading- und Eviction-Strategien).

Beim Speichern eines Werts kümmert sich der Schreiber darum, dass die Änderung im Backend-Speicher festgehalten wird. Nimmt zum Beispiel node1 Änderung C1 und node2 C2 vor, so teilt node1 seinem Cache Loader mit, dass C1 gespeichert werden soll, während node2 seinen Cache Loader mit der Speicherung von C2 beauftragt.

9.5.3. Replizierte Caches, darunter nur ein Knoten mit Speicher

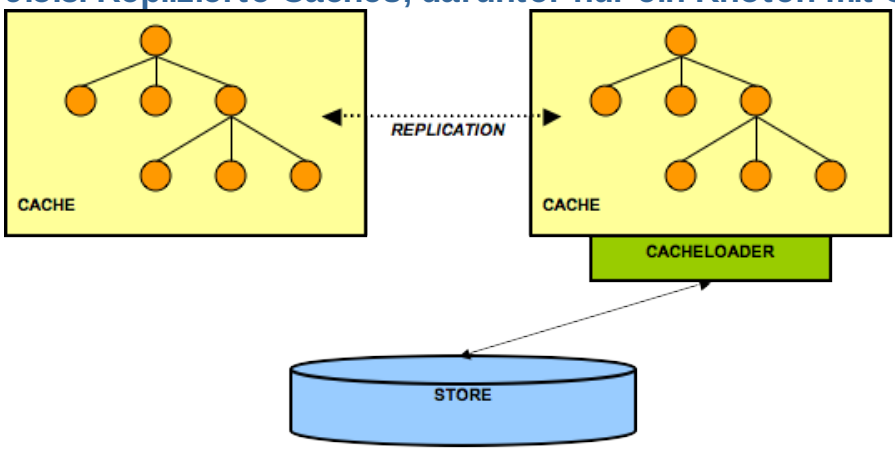


Abbildung 9.3. Zwei Knoten, von denen jedoch nur einer auf den Backend-Speicher zugreift

Dies ähnelt dem vorherigen Fall, hier allerdings interagiert nur ein Knoten im Cluster über seinen Cache Loader mit einem Backend-Speicher. Alle anderen Knoten führen speicherinterne Replikation durch. Die Idee hierbei ist, dass der gesamte Applikationszustand im Arbeitsspeicher eines jeden Knotens bewahrt wird, so dass durch das Vorhandensein mehrerer Caches diese Daten hochverfügbar werden. (Dies setzt voraus, dass ein Client, der die Daten benötigt, im Fehlerfall von einem Cache auf einen anderen ausweichen kann.) Der einzige persistente Backend-Speicher liefert eine Sicherungskopie der Daten, falls alle Caches im Cluster ausfallen oder neu gestartet werden müssen.

Beachten Sie, dass es bei dieser Konstellation sinnvoll sein kann, wenn der Cache Loader Änderungen asynchron speichert, also *nicht* im Anrufer-Thread, damit der Cluster nicht durch Zugreifen auf (zum Beispiel) eine Datenbank verlangsamt wird. Bei asynchroner Replikation spielt dies keine Rolle.

Der Schwachpunkt dieser Architektur besteht darin, dass mit Ausfall des einen Caches, der Zugriff auf den Cache Loader hat, auch das gesamte System ausfällt (dieser Cache ist ein sog. "single point of failure"). Außerdem muss bei einem Neustart des Clusters der Cache mit dem Cache Loader als erstes gestartet werden (dies kann leicht vergessen werden). Eine Lösung für das erste Problem wäre die Konfiguration eines Cache Loaders auf jedem Knoten, doch einer **`singletonStore`**-Einstellung auf **`true`**. Mit dieser Art der Konfiguration schreibt immer genau ein Knoten in den persistenten Speicher. Allerdings wird hierdurch das zweite Problem eher noch komplizierter, denn vor einem Neustart müssen Sie nun feststellen, welcher Cache vor dem Ausfall der schreibende Cache war, um diesen dann zuerst neu zu starten.

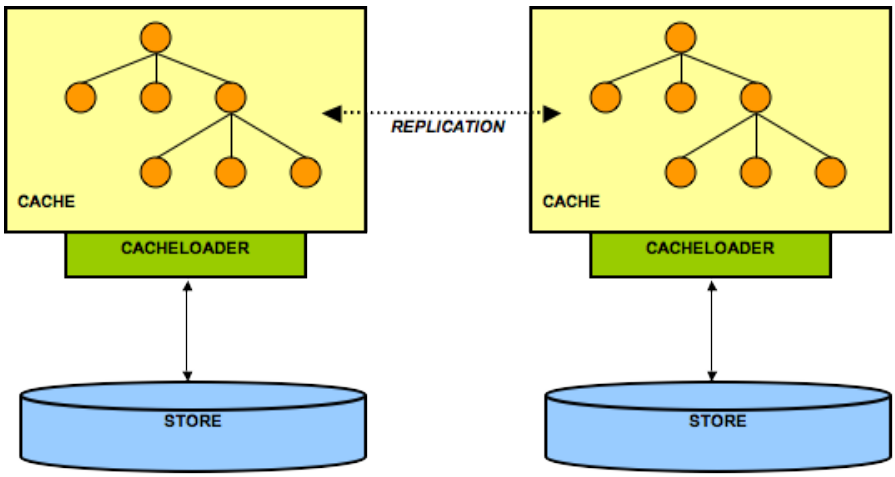


Abbildung 9.4. Zwei Knoten mit jeweils eigenem Backend-Speicher

Hier hat jeder Knoten seinen eigenen Datenspeicher. Änderungen am Cache werden (a) über den Cluster hinweg repliziert und (b) mittels Cache Loader persistiert. Das bedeutet, dass alle Datenspeicher exakt denselben Zustand besitzen. Bei der synchronen Replikation von Änderungen in einer Transaktion kümmert sich das Zwei-Phasen Festschreibungsprotokoll darum, dass alle Änderungen repliziert und in jedem Datenspeicher persistiert werden, oder aber nichts repliziert und persistiert wird (atomische Aktualisierung).

Beachten Sie, dass JBoss Cache derzeit *keine* XA-Ressource ist, was bedeutet, dass er keine Funktionalität zur Wiederherstellung ("Recovery") implementiert. Wird er zusammen mit einem Transaktionsmanager genutzt, der Recovery unterstützt, ist diese Funktionalität nicht verfügbar.

Die Herausforderung hier ist die Zustandsübertragung: Startet ein neuer Knoten, muss Folgendes geschehen:

1. Aufforderung an den Koordinator (den ältesten Knoten im Clusters), seinen Zustand zu senden. Hierbei handelt es sich immer um eine vollständige Zustandsübertragung, bei der stämtliche, eventuell bereits vorhandene Zustände überschrieben werden.
2. Der Koordinator muss dann warten, bis alle aktuell laufenden Transaktionen beendet sind. Während dieser Zeit gestattet er den Start neuer Transaktionen nicht.
3. Anschließend fragt der Koordinator mittels **`loadEntireState()`** seinen Cache Loader nach dem gesamten Zustand. Dieser Zustand wird dann an den neuen Knoten geschickt.
4. Der neue Knoten teilt seinem Cache Loader dann mit, dass der Zustand in seinem Speicher gespeichert werden soll, wobei der alte Zustand überschrieben wird. Dies ist die **`CacheLoader.storeEntireState()`**-Methode.
5. Optional kann auch der transiente (speicherinterne) Zustand während der Zustandsübertragung übertragen werden.
6. Der neue Knoten besitzt jetzt denselben Zustand in seinem Backend-Speicher wie alle anderen im Cluster, und von anderen Knoten erhaltene Änderungen werden jetzt unter Verwendung des lokalen Cache Loaders persistiert.

9.5.5. Hierarchische Caches

Falls Sie innerhalb einer einzelnen VM eine Hierarchie aufstellen müssen, so können Sie den **`LocalDelegatingCacheLoader`** verwenden. Dieser Hierarchietyp kann derzeit nur befehlsorientiert eingestellt werden.

Hierarchische Caches können sich mit Hilfe des **`TcpDelegatingCacheLoader`** auch über mehr als

eine JVM oder einen Server erstrecken.

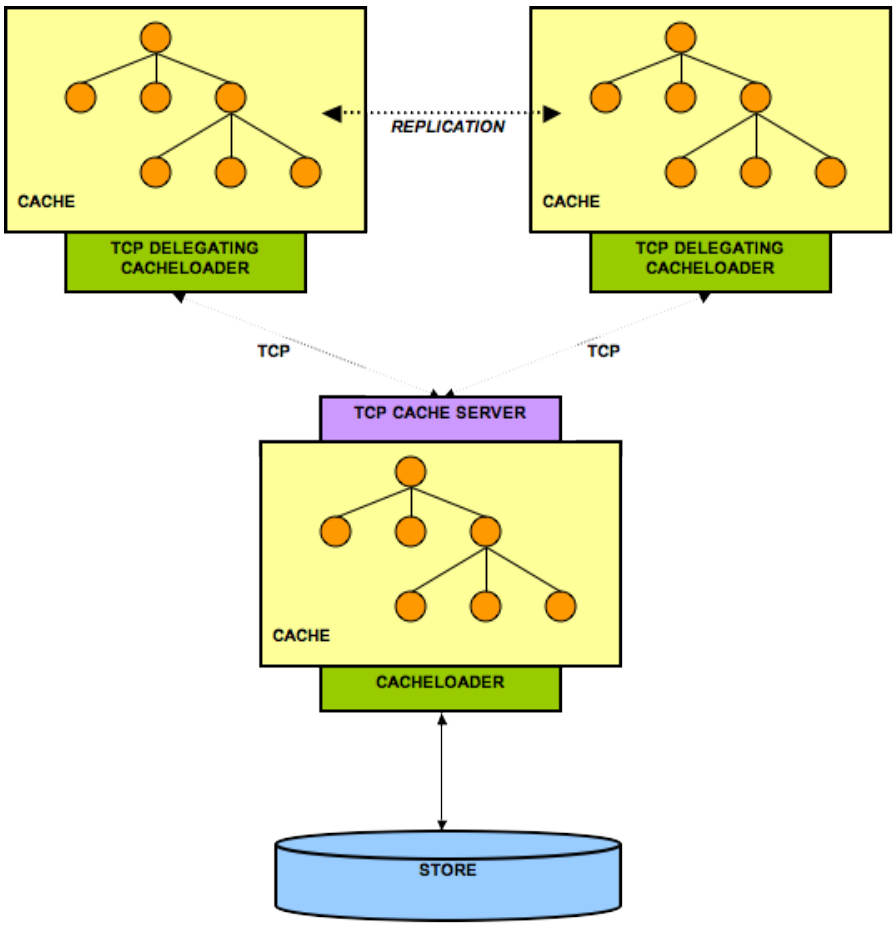


Abbildung 9.5. TCP delegierender Cache Loader

9.5.6. Mehrere Cache Loader

Sie können mehr als einen Cache Loader in einer Kette hintereinanderschalten. Intern wird ein delegierender **`ChainingCacheLoader`** verwendet, mit Referenzen auf jeden konfigurierten Cache Loader. Es gibt hierfür verschiedene Anwendungsfälle, abhängig von der Art der in der Kette eingesetzen Cache Loader. Ein Beispiel ist die Verwendung eines dateibasierten Cache Loaders, der sich auf demselben Host wie die JVM befindet und als Speicherüberlauf verwendet wird. Dies gewährleistet, dass Daten einfach verfügbar sind und das zu geringen Kosten. Ein zusätzlicher, entfernter Cache Loader, wie z.B. ein **`TcpDelegatingCacheLoader`** bietet Ausfallsicherheit zwischen Server-Neustarts.

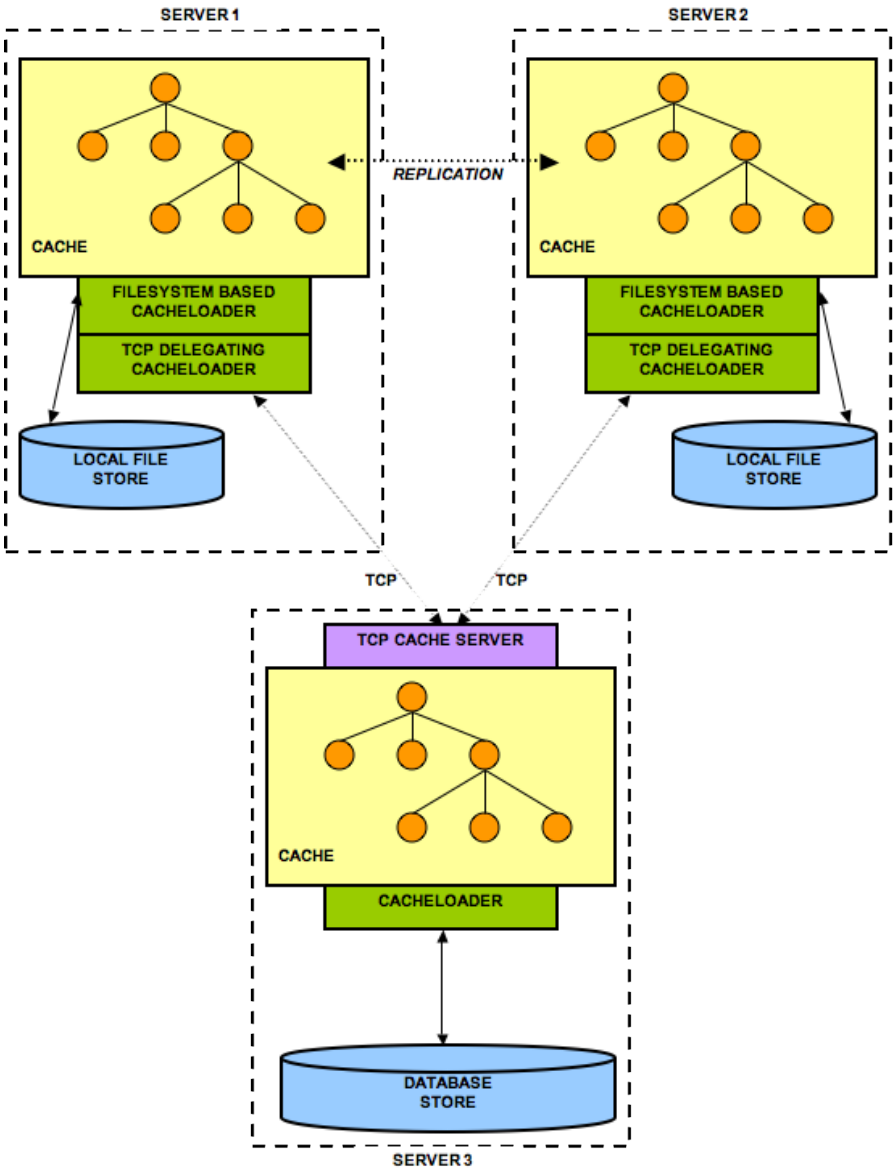


Abbildung 9.6. Mehrere Cache Loader in einer Kette

Kapitel 10. Eviction

Eviction (etwa "Räumung") steuert die Speicherverwaltung von JBoss Cache, indem die Anzahl und die Verweildauer der Knoten im Speicher beschränkt wird. Begrenzter Speicher auf Servern bedeutet, dass Caches nicht unbegrenzt wachsen können. Eviction ist deshalb notwendig, um Fehler aufgrund mangelnden Speicherplatzes zu vermeiden. Eviction wird meist zusammen mit Cache Loadern (siehe Kapitel 9, *Cache Loader*) verwendet.

10.1. Aufbau

Eviction in JBoss Cache baut auf vier Aspekten auf:

1. Sammeln statistischer Daten
2. Bestimmen der zu räumenden Knoten
3. Art und Weise der Knotenräumung
4. Eviction-Threads

Außerdem spielen Bereiche eine Schlüsselrolle bei der Eviction, da Eviction immer pro Bereich konfiguriert wird, so dass verschiedene Unterbäume im Cache unterschiedliche Eviction-Charakteristiken haben können.

10.1.1. Sammeln statistischer Daten

Dies erfolgt auf dem aufrufenden Thread, wann immer jemand mit dem Cache interagiert. Ist Eviction aktiviert, wird ein **`EvictionInterceptor`** zur Interzeptorkette hinzugefügt und Ereignisse werden in einer Ereignis-Queue aufgezeichnet. Ereignisse sind durch die **`EvictionEvent`**-Klasse gekennzeichnet. Ereignis-Queues werden für spezifische Bereiche geführt, so dass jeder Bereich über seine eigene Ereignis-Queue verfügt.

Dieser Aspekt der Eviction ist über das Hinzufügen oder Weglassen des **`EvictionInterceptor`** in der Interzeptorkette hinaus (abhängig davon, ob Eviction aktiviert ist) nicht konfigurierbar.

10.1.2. Bestimmen der zu räumenden Knoten

Eine **`EvictionAlgorithm`**-Implementierung verarbeitet die Eviction Queue um zu entscheiden, welche Knoten geräumt werden sollen. JBoss Cache enthält eine Reihe dieser Implementierungen, u.a. **`FIFOAlgorithm`**, **`LRUAlgorithm`**, **`LFUAlgorithm`**, etc. Jede Implementierung hat eine entsprechende **`EvictionAlgorithmConfig`**-Implementierung mit Konfigurationsdetails für den Algorithmus.

Angepasste **`EvictionAlgorithm`**-Implementierungen können durch Implementierung der Schnittstelle oder Erweitern einer der gelieferten Implementierungen bereitgestellt werden.

Algorithmen werden ausgeführt durch Aufruf ihrer **`process()`**-Methode und Übergabe der Ereignis-Queue zur Verarbeitung. In der Regel erfolgt dies durch Aufruf von **`Region.processEvictionQueues()`**, was den diesem Bereich zugewiesenen Algorithmus abruft.

10.1.3. Art und Weise der Knotenräumung

Sobald der **`EvictionAlgorithm`** entschieden hat, welche Knoten geräumt werden sollen, verwendet er eine Implementierung von **`EvictionActionPolicy`** um festzustellen, auf welche Weise die Knoten zu räumen sind. Dies ist pro Bereich konfigurierbar und ist standardmäßig **`DefaultEvictionActionPolicy`**, wodurch für jeden zu räumenden Knoten **`Cache.evict()`** aufgerufen wird.

JBoss Cache enthält auch eine **`RemoveOnEvictActionPolicy`**, wodurch für jeden zu räumenden Knoten **`Cache.removeNode()`** aufgerufen wird statt **`Cache.evict()`**.

Angepasste **`EvictionActionPolicy`**-Implementierungen können ebenfalls eingesetzt werden.

10.1.4. Eviction-Threads

Standardmäßig wird ein einziger, Cache-übergreifender Eviction-Thread verwendet, um regelmäßig registrierte Bereiche durchzugehen und **`Region.processEvictionQueues()`** auf jedem Bereich aufzurufen. Die Häufigkeit, mit der dieser Thread ausgeführt wird, kann mit Hilfe des **`wakeUpInterval`**-Attributs im **`eviction`**-Konfigurationselement konfiguriert werden; sie beträgt standardmäßig 5000 Millisekunden, falls nicht abweichend spezifiziert.

Der Eviction-Thread kann deaktiviert werden, indem Sie **`wakeUpInterval`** auf **`0`** setzen. Dies kann nützlich sein, wenn Sie Ihren eigenen regelmäßig laufenden Wartungs-Thread haben und damit selbst die Regionen durchgehen und **`Region.processEvictionQueues()`** aufrufen möchten.

10.2. Eviction-Bereiche

Das Konzept der Bereiche (auch: Regionen) und der **`Region`**-Klasse wurden in Abschnitt 7.6, „Klassenladen und Bereiche“ im Zusammenhang mit Marshalling erwähnt. Bereiche werden auch dazu genutzt, um das Eviction-Verhalten für Knoten innerhalb dieses Bereichs festzulegen. Zusätzlich zu der bereichsspezifischen Konfiguration können Sie auch ein standardmäßiges, Cache-weites Eviction-Verhalten für alle Knoten festlegen, die nicht in vordefinierte Bereiche fallen, oder falls Sie keine spezifischen Regionen definieren möchten. Beachten Sie, dass beim Definieren von Bereichen mittels der Konfigurations-XML-Datei alle Elemente des **`Fqn`**, der den Bereich definiert, **`String`**-Objekte sind.

Für jeden Bereich können Sie Eviction-Parameter festlegen.

Es ist möglich, Bereiche zu definieren, die sich überschneiden. Zum Beispiel kann ein Bereich für **`/a/b/c`** definiert sein, während ein anderer für **`/a/b/c/d`** definiert ist (also nur der *d* Unterbaum des **`/a/b/c`** Unterbaums). Der Algorithmus, um ein solches Szenario konsistent zu handhaben, wird immer den ersten Bereich wählen, den er antrifft. Auf diese Weise würde ein Algorithmus, der entscheiden muss, wie mit dem Knoten **`/a/b/c/d/e`** zu verfahren ist, von dort starten und im Baum aufwärts steigen, bis er den ersten definierten Bereich antrifft – in diesem Fall **`/a/b/c/d`**.

10.2.1. Residente Knoten

Als resident gekennzeichnete Knoten (mittels der **`Node.setResident()`**-API) werden von den Eviction-Richtlinien sowohl beim Prüfen, ob die Eviction ausgelöst werden soll, als auch beim Durchführen der eigentlichen Räumung von Knoten ignoriert. Wenn beispielsweise ein Bereich für maximal zehn Knoten konfiguriert ist, werden residente Knoten für die Entscheidung, ob Knoten aus diesem Bereich geräumt werden müssen, nicht mitgezählt. Außerdem werden residente Knoten nicht für die Räumung in Betracht gezogen, wenn die Grenze erreicht ist.

Um einen Knoten als resident zu kennzeichnen, sollte die **`Node.setResident()`**-API verwendet werden. Standardmäßig sind die neu erzeugten Knoten nicht resident. Das **`resident`**-Attribut eines Knotens wird nicht repliziert oder persistiert und ist nicht transaktionsfähig.

Als Beispiel für einen Anwendungsfall residenter Knoten könnte auf diese Weise sichergestellt werden, dass Pfadknoten keinen zusätzlichen "Lärm" in einer Eviction-Richtlinie verursachen, z.B.:

```
...
   Map lotsOfData = generateData();
   cache.put("/a/b/c", lotsOfData);
   cache.getRoot().getChild("/a").setResident(true);
   cache.getRoot().getChild("/a/b").setResident(true);
...
```

In diesem Beispiel sind die Knoten **/a** und **/a/b** Pfade, die nur vorhanden sind, um die Existenz des Knotens **/a/b/c** zu ermöglichen, und sie enthalten selbst keinerlei Daten. Als solches sind sie gute Kandidaten zur Kennzeichnung als residente Knoten. Dadurch wird die Speicherverwaltung verbessert, da keine Eviction-Ereignisse generiert werden müssen, wenn auf **/a** und**/a/b** zugegriffen wird.

Anmerkung

Wenn Attribute zu einem residenten Knoten hinzugefügt werden, z.B. **`cache.put("/a", "k", "v")`** im obigen Beispiel, macht es Sinn, die Knoten wieder als nicht resident zu kennzeichnen, damit diese nunmehr bei der Eviction berücksichtigt werden.

10.3. Konfiguration der Eviction

10.3.1. Grundlegende Konfiguration

Das grundlegende Eviction-Konfigurationselement sieht folgendermaßen aus:

```
   ...
   <eviction wakeUpInterval="500" eventQueueSize="100000">
      <default algorithmClass="org.jboss.cache.eviction.LRUAlgorithm">
         <property name="maxNodes" value="5000" />
         <property name="timeToLive" value="1000" />
      </default>
   </eviction>
   ...
```

- **`wakeUpInterval`** – dieser erforderliche Parameter legt in Millisekunden fest, wie oft der Eviction-Thread ausgeführt werden soll.
- **`eventQueueSize`** – dieser optionale Parameter legt die Größe der Queue fest, die Eviction-Ereignisse enthält. Wenn Ihr Eviction-Thread nicht häufig genug läuft, kann sich u.U. die Ereignis-Queue füllen. In diesem Fall kann es nötig sein, Ihren Eviction-Thread häufiger auszuführen, oder Ihre Ereignis-Queue zu vergrößern. Diese Konfiguration ist nur die *standardmäßige* Größe der Ereignis-Queue und kann in einzelnen Eviction-Bereichen außer Kraft gesetzt werden. Falls nicht spezifiziert, lautet der Standard **`200000`**.
- **`algorithmClass`** – dies ist erforderlich, es sei denn, Sie legen individuelle **`algorithmClass`**-Attribute für jeden einzelnen Bereich fest. Dies definiert den standardmäßigen Eviction-Algorithmus, der verwendet werden soll, falls keiner gesondert für den Bereich definiert ist.
- Algorithmus-Konfigurationsattribute – diese sind spezifisch für den in **`algorithmClass`** spezifizierten Algorithmus. Werfen Sie für weitere Details bitte einen Blick auf den Abschnitt über den jeweiligen Algorithmus.

10.3.2. Befehlsorientierte Konfiguration

Die Konfiguration der Eviction mit Hilfe des **`Configuration`**-Objekts erfordert die Verwendung der

`org.jboss.cache.config.EvictionConfig`-Bean, die an **`Configuration.setEvictionConfig()`** übergeben wird. Siehe Kapitel 3, *Konfiguration* für weitere Informationen über das befehlsorientierte Erzeugen einer **`Configuration`**.

Durch die Verwendung einfacher POJO-Beans, um alle Elemente in einer Cache-Konfiguration zu repräsentieren, ist es auch relativ einfach, nach dem Start des Caches noch befehlsorientiert Eviction-Bereiche hinzuzufügen. Nehmen wir beispielsweise an, wir haben einen vorhandenen Cache, konfiguriert mittels XML mit dem oben gezeigten Eviction-Konfigurationselement. Nun möchten wir während der Laufzeit einen neuen Eviction-Bereich namens "/org/jboss/fifo" hinzufügen, unter Verwendung von **`LRUAlgorithm`**, aber einer anderen Anzahl von **`maxNodes`**:

```
Fqn fqn = Fqn.fromString("/org/jboss/fifo");

// Create a configuration for an LRUPolicy
LRUAlgorithmConfig lruc = new LRUAlgorithmConfig();
lruc.setMaxNodes(10000);

// Create an eviction region config
EvictionRegionConfig erc = new EvictionRegionConfig(fqn, lruc);

// Create the region and set the config
Region region = cache.getRegion(fqn, true);
region.setEvictionRegionConfig(erc);
```

10.4. Enthaltene Eviction-Richtlinien

Dieser Abschnitt erläutert die verschiedenen, in JBoss Cache enthaltenen Algorithmen sowie die verschiedenen Konfigurationsparameter für jeden Algorithmus.

10.4.1. LRUAlgorithm – Least Recently Used ("Am längsten nicht verwendet")

`org.jboss.cache.eviction.LRUAlgorithm` steuert sowohl die Lebensdauer als auch das Alter der Knoten. Diese Richtlinie garantiert eine konstante Reihenfolge (**`O (1)`**) für Hinzufügungen, Entfernungen und Lookups (Aufrufe). Sie besitzt folgende Konfigurationsparameter:

- **`maxNodes`** – Dies ist die maximale in diesem Bereich gestattete Anzahl von Knoten. 0 bedeutet sofortigen Ablauf, -1 bedeutet keine Beschränkung.
- **`timeToLive`** – Zeit ohne Schreib- oder Lesevorgänge im Knoten (in Millisekunden), ehe der Knoten bereinigt wird. 0 bedeutet sofortigen Ablauf, -1 bedeutet keine Beschränkung.
- **`maxAge`** – Lebensdauer eines Knotens (in Millisekunden) nach der, ungeachtet der Dauer der Untätigkeit, der Knoten bereinigt wird. 0 bedeutet sofortigen Ablauf, -1 bedeutet keine Beschränkung.
- **`minTimeToLive`** – die Mindestzeit, die ein Knoten nach Zugriff darauf bestehen bleiben darf, bevor er für die Eviction in Betracht gezogen werden darf. 0 bedeutet, dass diese Funktion deaktiviert ist, dies ist die Standardeinstellung.

10.4.2. FIFOAlgorithm - First In, First Out ("Als Erster rein, als Erster raus")

`org.jboss.cache.eviction.FIFOAlgorithm` steuert die Eviction in ordnungsgemäßer "First In, First Out" Reihenfolge (etwa "Als Erster rein, als Erster raus"). Diese Richtlinie garantiert eine konstante Reihenfolge (**`O (1)`**) für Hinzufügungen, Entfernungen und Lookups (Aufrufe). Sie besitzt folgende Konfigurationsparameter:

- **`maxNodes`** – Dies ist die maximale in diesem Bereich gestattete Anzahl von Knoten. 0 bedeutet sofortigen Ablauf, -1 bedeutet keine Beschränkung.

- **`minTimeToLive`** – die Mindestzeit, die ein Knoten nach Zugriff darauf bestehen bleiben darf, bevor er für die Eviction in Betracht gezogen werden darf. 0 bedeutet, dass diese Funktion deaktiviert ist, dies ist die Standardeinstellung.

10.4.3. MRUAlgorithm - Most Recently Used ("Zuletzt verwendet")

`org.jboss.cache.eviction.MRUAlgorithm` steuert die Eviction basierend auf dem "Most Recently Used" ("Zuletzt verwendet") Algorithmus. Die zuletzt verwendeten Knoten werden mit dieser Richtlinie als Erste geräumt. Diese Richtlinie garantiert eine konstante Reihenfolge (**`O (1)`**) für Hinzufügungen, Entfernungen und Lookups (Aufrufe). Sie besitzt folgende Konfigurationsparameter:

- **`maxNodes`** – Dies ist die maximale in diesem Bereich gestattete Anzahl von Knoten. 0 bedeutet sofortigen Ablauf, -1 bedeutet keine Beschränkung.
- **`minTimeToLive`** – die Mindestzeit, die ein Knoten nach Zugriff darauf bestehen bleiben darf, bevor er für die Eviction in Betracht gezogen werden darf. 0 bedeutet, dass diese Funktion deaktiviert ist, dies ist die Standardeinstellung.

10.4.4. LFUAlgorithm - Least Frequently Used ("Am wenigsten verwendet")

`org.jboss.cache.eviction.LFUAlgorithm` steuert die Eviction basierend auf dem "Least Frequently Used" ("Am wenigsten verwendet") Algorithmus. Die am wenigsten verwendeten Knoten werden mit dieser Richtlinie als erste geräumt. Die Zählung der Knotenverwendung startet bei 1, wenn ein Knoten erstmals hinzugefügt wird. Bei jedem Zugriff erhöht sich der Knotenverwendungszähler um 1. Diese Zahl wird dann zur Bestimmung der am seltensten verwendeten Knoten herangezogen. LFU ist auch ein sortierter Eviction-Algorithmus. Die zugrunde liegende Eviction-Queue-Implementierung und der Algorithmus werden in aufsteigender Reihenfolge des Knotenverwendungszählers sortiert. Diese Klasse garantiert eine konstante Reihenfolge (**`O (1)`**) für Hinzufügungen, Entfernungen und Lookups (Aufrufe). Wird jedoch eine Anzahl von Knoten der Queue hinzugefügt oder für einen bestimmten Bearbeitungsgang verwendet, wird eine einzelne quasilineare (**`O (n * log n)`**) Operation verwendet, um die Queue wieder in korrekter LFU-Reihenfolge zu sortieren. Auf ähnliche Weise ist bei Entfernung oder Räumung von Knoten eine einzelne lineare (**`O (n)`**) Pruning-Operation zur Bereinigung der Eviction Queue nötig. LFU besitzt folgende Konfigurationsparameter:

- **`maxNodes`** – Dies ist die maximale in diesem Bereich gestattete Anzahl von Knoten. 0 bedeutet sofortigen Ablauf, -1 bedeutet keine Beschränkung.
- **`minNodes`** – Dies ist die Mindestanzahl von Knoten, die in diesem Bereich gestattet sind. Dieser Wert bestimmt, ob die Eviction Queue zurückgesetzt werden sollte. Ist minNodes zum Beispiel 10 und der Cache wächst auf 100 Knoten an, so wird der Cache auf die 10 am häufigsten verwendeten Knoten zurückgesetzt, wenn der Eviction-Timer den Eviction-Algorithmus durchläuft.
- **`minTimeToLive`** – die Mindestzeit, die ein Knoten nach Zugriff darauf bestehen bleiben darf, bevor er für die Eviction in Betracht gezogen werden darf. 0 bedeutet, dass diese Funktion deaktiviert ist, dies ist die Standardeinstellung.

10.4.5. ExpirationAlgorithm

`org.jboss.cache.eviction.ExpirationAlgorithm` ist eine Richtlinie, die Knoten basierend auf einer absoluten Verfallszeit räumt. Die Verfallszeit wird mit Hilfe der **`org.jboss.cache.Node.put()`**-Methode angegeben, mittels eines String-Schlüssels **`expiration`** und der absoluten Zeit als **`java.lang.Long`**-Objekt, mit einem Wert von Millisekunden nach Mitternacht am 1. Januar 1970 UTC (dieselbe relative Zeit wie von **`java.lang.System.currentTimeMillis()`** ausgegeben).

Diese Richtlinie garantiert eine konstante Reihenfolge (**`O (1)`**) für Hinzufügungen, Entfernungen und Lookups (Aufrufe). Intern wird ein sortiertes Set (TreeSet) gespeichert, das die Verfallszeit und den FQN der Knoten enthält und im Grunde genommen als ein Heap fungiert.

Diese Richtlinie besitzt folgende Konfigurationsparameter:

- **`expirationKeyName`** – Dies ist der Knoten-Schlüsselname, der im Eviction-Algorithmus verwendet wird. Der Konfigurationsstandard lautet **`expiration`**.
- **`maxNodes`** – Dies ist die maximale in diesem Bereich gestattete Anzahl von Knoten. 0 bedeutet sofortigen Ablauf, -1 bedeutet keine Beschränkung.

Das folgende Listing zeigt, wie das Verfallsdatum gekennzeichnet wird und wie die Richtlinie angewendet wird:

```
   Cache cache = DefaultCacheFactory.createCache();
   Fqn fqn1 = Fqn.fromString("/node/1");
   Long future = new Long(System.currentTimeMillis() + 2000);

   // sets the expiry time for a node
   cache.getRoot().addChild(fqn1).put(ExpirationConfiguration.EXPIRATION_KEY,
future);

   assertTrue(cache.getRoot().hasChild(fqn1));
   Thread.sleep(5000);

   // after 5 seconds, expiration completes
   assertFalse(cache.getRoot().hasChild(fqn1));
```

Beachten Sie, dass die Verfallszeit von Knoten nur überprüft wird, wenn der Bereichsmanager alle in **`wakeUpIntervalSeconds`** festgelegten Sekunden aufwacht, die Eviction kann also ein paar Sekunden später als angegeben erfolgen.

10.4.6. ElementSizeAlgorithm – Eviction basierend auf der Anzahl von Schlüssel/Wert-Paaren in einem Knoten

`org.jboss.cache.eviction.ElementSizeAlgorithm` steuert die Eviction basierend auf der Anzahl von Schlüssel/Wert-Paaren in einem Knoten. Die zuletzt verwendeten Knoten werden mit dieser Richtlinie als erste geräumt. Sie besitzt folgende Konfigurationsparameter:

- **`maxNodes`** – Dies ist die maximale in diesem Bereich gestattete Anzahl von Knoten. 0 bedeutet sofortigen Ablauf, -1 bedeutet keine Beschränkung.
- **`maxElementsPerNode`** – Dies ist die auslösende Anzahl von Attributen pro Knoten, bei der dieser Knoten zur Eviction ausgewählt wird. 0 bedeutet sofortigen Ablauf, -1 bedeutet keine Beschränkung.
- **`minTimeToLive`** – die Mindestzeit, die ein Knoten nach Zugriff darauf bestehen bleiben darf, bevor er für die Eviction in Betracht gezogen werden darf. 0 bedeutet, dass diese Funktion deaktiviert ist, dies ist die Standardeinstellung.

Kapitel 11. Transaktionen und Nebenläufigkeit

11.1. Nebenläufiger Zugriff

JBoss Cache ist eine threadsichere Caching-API und verwendet seine eigenen, effizienten Verfahren zur Steuerung von nebenläufigen Zugriffen. Es verwendet eine innovative Implementierung von Multi-Versioned Concurrency Control (MVCC) als standardmäßiges Sperrschema. Versionen von JBoss Cache vor 3.x boten optimistische und pessimistische Sperrschemata an, die nunmehr jedoch beide zugunsten von MVCC aufgegeben wurden.

11.1.1. Multi-Version Concurrency Control (MVCC)

MVCC ist ein Sperrschema, das üblicherweise in modernen Datenbankimplementierungen eingesetzt wird, um den schnellen, sicheren, nebenläufigen Zugriff auf gemeinsam verwendete Daten zu steuern.

11.1.1.1. MVCC-Konzepte

MVCC ist darauf ausgelegt, die folgenden Features für nebenläufigen Zugriff zu bieten:

- Lesevorgänge blockieren keine Schreibvorgänge
- Schreibvorgänge, die schnell fehlschlagen

Erreicht wird dies mit Hilfe von Datenversionierung und Kopieren für nebenläufige Schreibvorgänge. Die Theorie dabei ist, dass Lesevorgänge weiterhin die freigegebenen Zustände lesen, während Schreibvorgänge die freigegebenen Zustände kopieren, eine Versions-ID erhöhen, und anschließend den Zustand zurückschreiben, nachdem sichergestellt wurde, dass die Version noch gültig ist (d.h. kein anderer, nebenläufiger Schreibvorgang hat den Zustand in der Zwischenzeit verändert).

Dies erlaubt es Lesevorgängen, weiterhin zu lesen, während Schreibvorgänge nicht am Schreiben gehindert werden, und wiederholbare Lesesemantiken werden bewahrt, indem Lesern ermöglicht wird, die alte Version des Zustands zu lesen.

11.1.1.2. MVCC-Implementierung

JBoss Cache's Implementierung von MVCC basiert auf einigen wenigen Features:

- Lesevorgänge erhalten keine Sperren
- Nur eine zusätzliche Version wird für den gemeinsam genutzten Zustand bewahrt, für einen einzigen Schreibvorgang
- Alle Schreibvorgänge erfolgen sequenziell, um schnelle Ausfall-Semantiken zu bieten

Die extrem hohe Leistung von JBoss Cache's MVCC-Implementierung für lesende Threads wird erreicht, da für Lesevorgänge keine Synchronisation oder Sperre erforderlich ist. Für jeden lesenden Thread hüllt der **`MVCCLockingInterceptor`** den Zustand in ein schlankes Containerobjekt, das in den **`InvocationContext`** (oder den **`TransactionContext`**, falls in einer Transaktion) des Threads platziert wird. Alle nachfolgenden Operationen auf dem Zustand finden über das Containerobjekt statt. Ein derartiger Einsatz von Java-Referenzen ermöglicht wiederholbare Lese-Semantiken, selbst wenn sich der eigentliche Zustand gleichzeitig verändert.

Schreibende Threads hingegen müssen eine Sperre beziehen, bevor der Schreibvorgang beginnen kann. Derzeit verwenden wir "Lock Striping", um die Speicherleistung des Caches zu verbessern. Die Größe des gemeinsam verwendeten Sperr-Pools kann mit Hilfe des **`concurrencyLevel`**-Attributs des **`locking`**-Elements optimiert werden. Werfen Sie für weitere Einzelheiten einen Blick auf das Kapitel 12, *Konfigurationsreferenzen*. Nach Erhalt einer exklusiven Sperre auf einem Fqn hüllt der Schreibvorgang dann den zu verändernden Zustand ebenfalls in einen Container, ganz wie auch die Lese-Threads, und

kopiert anschließend diesen Zustand zum Schreiben. Beim Kopieren wird im Container eine Referenz zur Originalversion bewahrt (für eventuelle Rollbacks, das Zurücksetzen des Zustands). Änderungen werden dann an der Kopie vorgenommen, und sobald der Schreibvorgang abgeschlossen ist, wird die Kopie schließlich wieder in die Datenstruktur zurückgeschrieben.

Auf diese Weise sehen nachfolgende Leser die neue Version, während derzeitige Leser noch immer eine Referenz auf die Originalversion in ihrem Kontext besitzen.

Kann ein Schreibvorgang nach einer gewissen Zeit immer noch keine Schreibsperre erhalten, wird eine **`TimeoutException`** geworfen. Diese Zeitüberschreitung beim Erhalt der Sperre ist standardmäßig 10000 Millisekunden und kann mit Hilfe des **`lockAcquisitionTimeout`**-Attributs des **`locking`**-Elements konfiguriert werden. Siehe Kapitel 12, *Konfigurationsreferenzen* für Details.

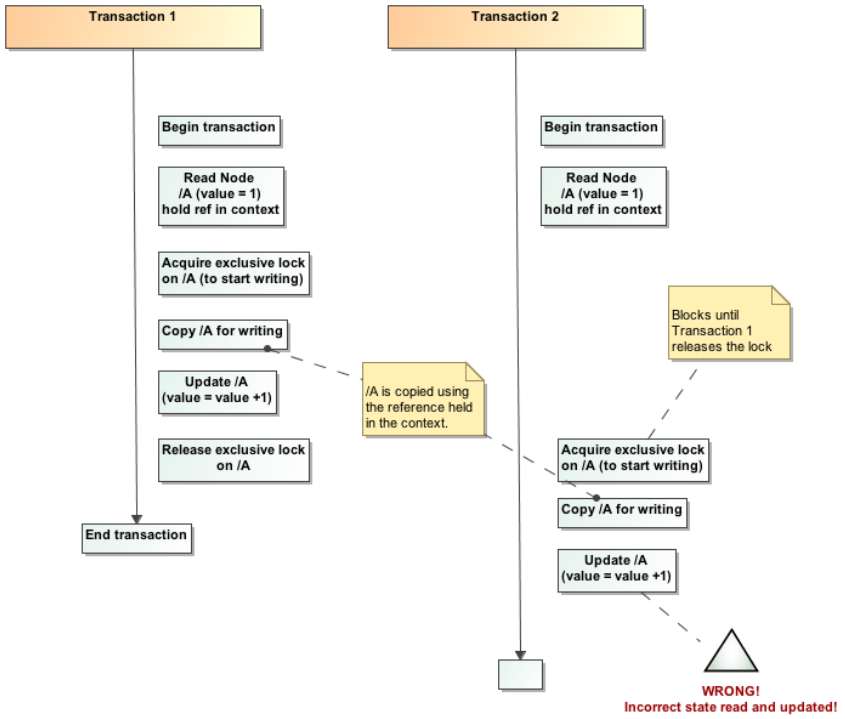
11.1.1.2.1. Isolationsebenen

JBoss Cache 3.x unterstützt zwei Isolationsebenen: REPEATABLE_READ und READ_COMMITTED, die semantisch datenbankartigen Isolationsebenen entsprechen. Frühere Versionen von JBoss Cache unterstützten alle fünf Datenbank-Isolationsebenen, und wenn eine nicht unterstützte Isolationsebene konfiguriert wird, wird diese auf die nächsthöhere oder -niedrigere Ebene angepasst.

REPEATABLE_READ ist die standardmäßige Isolationsebene, um die Kompatibilität mit älteren Versionen von JBoss Cache zu bewahren. READ_COMMITTED bietet zwar nur eine geringfügig schwächere Isolation, dafür jedoch eine deutlich bessere Leistung im Vergleich zu REPEATABLE_READ.

11.1.1.2.2. Nebenläufige Schreibvorgänge und Write-Skews

Obwohl MVCC Schreibvorgänge dazu zwingt, eine Schreibsperre zu beziehen, kann es beim Einsatz von REPEATABLE_READ dennoch zu einem Phänomen namens "Write Skews" kommen:


Transaction 1
Transaction 2
Begin transaction
Begin transaction
Read Node /A (value = 1) hold ref in context
Read Node /A (value = 1) hold ref in context
Acquire exclusive lock on /A (to start writing)
Blocks until Transaction 1 releases the lock
Copy /A for writing
/A is copied using the reference held in the context.
Update /A (value = value +1)
Release exclusive lock on /A
Acquire exclusive lock on /A (to start writing)
End transaction
Copy /A for writing
Update /A (value = value +1)
WRONG!
Incorrect state read and updated!

Dies passiert, wenn nebenläufige Transaktionen zunächst einen Lese- und anschließend einen Schreibvorgang ausführen, basierend auf dem gelesenen Wert. Da für Lesevorgänge eine Referenz auf den Zustand im Transaktionskontext bewahrt wird, wird ein nachfolgender Schreibvorgang auf Grundlage dieses gelesenen Originalzustands durchgeführt, der mittlerweile jedoch unter Umständen nicht mehr gültig ist.

Standardmäßig wird bei Auftreten eines Write Skews eine **`DataVersioningException`** ausgegeben, sobald der Fehler beim Kopieren des Zustands für das Schreiben entdeckt wird. Allerdings stellt ein Write Skew in den meisten Applikationen kein Problem dar (z.B. falls der geschriebene Zustand in keiner Beziehung zum gelesenen Originalzustand steht) und sollte erlaubt werden. Falls Ihrer Applikation Write Skews egal sind, können Sie diese erlauben, indem Sie das **`writeSkewCheck`**-Konfigurationsattribut auf **`false`** setzen. Siehe Kapitel 12, *Konfigurationsreferenzen* für Details.

Beachten Sie, dass Write Skews beim Einsatz von READ_COMMITTED nicht auftreten können, da Threads immer mit festgeschriebenen Zuständen arbeiten.

11.1.1.3. Konfiguration von Sperren

Zur Konfiguration von MVCC wird das **`<locking />`**-Konfigurations-Tag verwendet, und zwar folgendermaßen:

```
<locking
   isolationLevel="REPEATABLE_READ"
   lockAcquisitionTimeout="10234"
   nodeLockingScheme="mvcc"
   writeSkewCheck="false"
   concurrencyLevel="1000" />
```

- **`nodeLockingScheme`** – das verwendete Knoten-Sperrschema. Standardmäßig MVCC, falls nicht anders angegeben. Veraltete Schemata wie **`pessimistisch`** oder **`optimistisch`** können zwar verwendet werden, davon wird jedoch abgeraten.
- **`isolationLevel`** – Transaktions-Isolationsebene. Standardmäßig REPEATABLE_READ, falls nicht anders angegeben.
- **`writeSkewCheck`** – standardmäßig **`true`**, falls nicht anders angegeben.
- **`concurrencyLevel`** – standardmäßig 500, falls nicht anders angegeben.
- **`lockAcquisitionTimeout`** – nur relevant für Schreibvorgänge, wenn MVCC eingesetzt wird. Standardmäßig 10000, falls nicht anders angegeben.

11.1.2. Pessimistische und optimistische Sperrschemata

Ab JBoss Cache 3.x wurde das pessimistische und optimistische Sperrschema zugunsten von MVCC (siehe Abschnitt 11.1.1, „Multi-Version Concurrency Control (MVCC)“) aufgegeben. Es wird empfohlen, in existierenden Applikationen diese veralteten Sperrschemata aufzugeben, da diese in zukünftigen Versionen irgendwann nicht mehr unterstützt werden.

Eine Dokumentation dieser veralteten Sperrschemata ist in diesem Benutzerhandbuch nicht enthalten, kann aber, falls nötig, in früheren Versionen dieses Handbuchs auf der JBoss Cache Website eingesehen werden.

11.2. JTA-Unterstützung

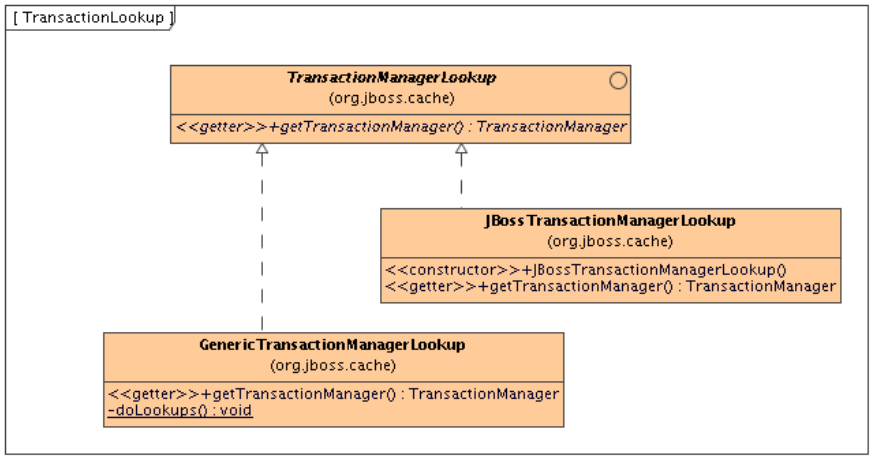
JBoss Cache kann so konfiguriert werden, dass es JTA-konforme Transaktionen unterstützt. Alternativ, wenn Transaktionsunterstützung deaktiviert ist, ist es gleichwertig zur AutoCommit-Einstellung, bei der potenziell nach jeder Änderung repliziert wird (falls Replikation aktiviert ist).

Bei jedem eingehenden Aufruf unternimmt JBoss Cache Folgendes:

1. Er bezieht die aktuelle **`javax.transaction.Transaction`**, die mit dem Thread verknüpft ist
2. Falls nicht schon erfolgt, registriert er eine **`javax.transaction.Synchronization`** beim Transaktionsmanager, um darüber benachrichtigt zu werden, wenn eine Transaktion

festgeschrieben oder zurückgesetzt wird.

Hierfür muss dem Cache eine Referenz zum **`javax.transaction.TransactionManager`** der Umgebung zur Verfügung gestellt werden. In der Regel geschieht dies, indem der Cache mit dem Klassennamen einer Implementierung der **`TransactionManagerLookup`**-Schnittstelle konfiguriert wird. Wenn der Cache startet, wird er eine Instanz dieser Klasse erzeugen und deren **`getTransactionManager()`**-Methode aufrufen, die mit einer Referenz auf den **`TransactionManager`** antwortet.



JBoss Cache wird mit **`JBossTransactionManagerLookup`** und **`GenericTransactionManagerLookup`** geliefert. Der **`JBossTransactionManagerLookup`** kann an eine laufende JBoss AS-Instanz anbinden und einen **`TransactionManager`** abrufen, während der **`GenericTransactionManagerLookup`** an die am häufigsten verwendeten Java EE Application Server anbinden und dieselbe Funktionalität bereitstellen kann. Eine Dummy-Implementierung – **`DummyTransactionManagerLookup`** – wird ebenfalls zum Testen geliefert. Wie der Name schon andeutet, wird diese Implementierung nicht für den Einsatz in Produktionsumgebungen empfohlen und hat einige schwerwiegende Einschränkungen hinsichtlich nebenläufiger Transaktionen und Recovery.

Als Alternative zur Konfiguration eines **`TransactionManagerLookup`** können Sie auch befehlsorientiert eine Referenz auf den **`TransactionManager`** in das **`RuntimeConfig`**-Element des **`Configuration`**-Objekts injizieren:

```
TransactionManager tm = getTransactionManager(); // magic method
cache.getConfiguration().getRuntimeConfig().setTransactionManager(tm);
```

Das Injizieren des **`TransactionManager`** ist sogar die empfohlene Vorgehensweise, wenn die **`Configuration`** von einer Art IOC-Container erzeugt wurde, der bereits eine Referenz auf den **`TransactionManager`** besitzt.

Wenn die Transaktion festschreibt, wird entweder das Ein-Phasen- oder das Zwei-Phasen-Festschreibungsprotokoll iniziiert. Siehe Abschnitt 8.1.2.1, „Replizierte Caches und Transaktionen“ für Einzelheiten.

Teil III. JBoss Cache Konfigurationsreferenzen

Dieser Abschnitt enhält technische Referenzen zum schnellen Nachschlagen.

Kapitel 12. Konfigurationsreferenzen

12.1. Beispiel einer XML-Konfigurationsdatei

Nachfolgend sehen Sie eine typische XML-Konfigurationsdatei. Wir empfehlen Ihnen, eine der mit der JBoss Cache Distribution mitgelieferten Konfigurationen als Ausgangspunkt für kleinere Anpassungen zu verwenden, statt eine Konfigurationsdatei ganz neu zu schreiben.

```
<?xml version="1.0" encoding="UTF-8"?>

<jbosscache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="urn:jboss:jbosscache-core:config:3.1">


   <!--
      isolation levels supported: READ_COMMITTED and REPEATABLE_READ
      nodeLockingSchemes: mvcc, pessimistic (deprecated), optimistic (deprecated)
   -->
   <locking
         isolationLevel="REPEATABLE_READ"
         lockParentForChildInsertRemove="false"
         lockAcquisitionTimeout="20000"
         nodeLockingScheme="mvcc"
         writeSkewCheck="false"
         useLockStriping="true"
         concurrencyLevel="500"/>

   <!--
      Used to register a transaction manager and participate in ongoing
transactions.
      -->
   <transaction

transactionManagerLookupClass="org.jboss.cache.transaction.GenericTransactionManagerL
ookup"
         syncRollbackPhase="false"
         syncCommitPhase="false"/>

   <!--
      Used to register JMX statistics in any available MBean server
      -->
   <jmxStatistics
         enabled="false"/>

   <!--
      If region based marshalling is used, defines whether new regions are
inactive on startup.
   -->
   <startup
         regionsInactiveOnStartup="true"/>

   <!--
      Used to register JVM shutdown hooks.
      hookBehavior: DEFAULT, REGISTER, DONT_REGISTER
   -->
   <shutdown
         hookBehavior="DEFAULT"/>

   <!--
      Used to define async listener notification thread pool size
   -->
   <listeners
         asyncPoolSize="1"
         asyncQueueSize="100000"/>

   <!--
      Used to enable invocation batching and allow the use of
Cache.startBatch()/endBatch() methods.
```

```
   -->
   <invocationBatching
         enabled="false"/>

   <!--
      serialization related configuration, used for replication and cache loading
   -->
   <serialization
         objectInputStreamPoolSize="12"
         objectOutputStreamPoolSize="14"
         version="3.0.0"
         marshallerClass="org.jboss.cache.marshall.VersionAwareMarshaller"
         useLazyDeserialization="false"
         useRegionBasedMarshalling="false"/>

   <!--
      This element specifies that the cache is clustered.
      modes supported: replication (r) or invalidation (i).
   -->
   <clustering mode="replication" clusterName="JBossCache-cluster">

      <!--
         Defines whether to retrieve state on startup
      -->
      <stateRetrieval timeout="20000" fetchInMemoryState="false"/>

      <!--
         Network calls are synchronous.
      -->
      <sync replTimeout="20000"/>
      <!--
         Uncomment this for async replication.
      -->
      <!--<async useReplQueue="true" replQueueInterval="10000"
replQueueMaxElements="500" serializationExecutorPoolSize="20"
serializationExecutorQueueSize="5000000"/>-->

      <!-- Uncomment to use Buddy Replication -->
      <!--
      <buddy enabled="true" poolName="myBuddyPoolReplicationGroup"
communicationTimeout="2000">
         <dataGravitation auto="true" removeOnFind="true"
searchBackupTrees="true"/>
         <locator class="org.jboss.cache.buddyreplication.NextMemberBuddyLocator">
            <properties>
               numBuddies = 1
               ignoreColocatedBuddies = true
            </properties>
         </locator>
      </buddy>
      -->

      <!--
         Configures the JGroups channel.  Looks up a JGroups config file on the
classpath or filesystem.  udp.xml
         ships with jgroups.jar and will be picked up by the class loader.
      -->
      <jgroupsConfig configFile="udp.xml">
         <!-- uncomment to define a JGroups stack here
```

```
         <PING timeout="2000" num_initial_members="3"/>
         <MERGE2 max_interval="30000" min_interval="10000"/>
         <FD_SOCK/>
         <FD timeout="10000" max_tries="5" shun="true"/>
         <VERIFY_SUSPECT timeout="1500"/>
         <pbcast.NAKACK use_mcast_xmit="false" gc_lag="0"
                        retransmit_timeout="300,600,1200,2400,4800"
                        discard_delivered_msgs="true"/>
         <UNICAST timeout="300,600,1200,2400,3600"/>
         <pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000"
                        max_bytes="400000"/>
         <pbcast.GMS print_local_addr="true" join_timeout="5000" shun="false"
                     view_bundling="true" view_ack_collection_timeout="5000"/>
         <FRAG2 frag_size="60000"/>
         <pbcast.STREAMING_STATE_TRANSFER use_reading_thread="true"/>
         <pbcast.FLUSH timeout="0"/>
          -->
      </jgroupsConfig>
   </clustering>

   <!--
      Eviction configuration.  WakeupInterval defines how often the eviction thread
runs, in milliseconds.  0 means
      the eviction thread will never run.
   -->
   <eviction wakeUpInterval="500">
      <default algorithmClass="org.jboss.cache.eviction.LRUAlgorithm"
eventQueueSize="200000">
         <property name="maxNodes" value="5000" />
         <property name="timeToLive" value="1000" />
      </default>
      <region name="/org/jboss/data1">
         <property name="timeToLive" value="2000" />
      </region>
      <region name="/org/jboss/data2"
algorithmClass="org.jboss.cache.eviction.FIFOAlgorithm" eventQueueSize="100000">
         <property name="maxNodes" value="3000" />
         <property name="minTimeToLive" value="4000" />
      </region>
   </eviction>

   <!--
      Cache loaders.

      If passivation is enabled, state is offloaded to the cache loaders ONLY when
evicted.  Similarly, when the state
      is accessed again, it is removed from the cache loader and loaded into
memory.

      Otherwise, state is always maintained in the cache loader as well as in
memory.

      Set 'shared' to true if all instances in the cluster use the same cache
loader instance, e.g., are talking to the
      same database.
   -->
   <loaders passivation="false" shared="false">
      <preload>
         <node fqn="/org/jboss"/>
         <node fqn="/org/tempdata"/>
```

```
      </preload>

      <!--
         we can have multiple cache loaders, which get chained
      -->
      <loader class="org.jboss.cache.loader.JDBCCacheLoader" async="true"
fetchPersistentState="true"
              ignoreModifications="true" purgeOnStartup="true">
         <properties>
            cache.jdbc.table.name=jbosscache
            cache.jdbc.table.create=true
            cache.jdbc.table.drop=true
         </properties>
         <singletonStore enabled="true"
class="org.jboss.cache.loader.SingletonStoreCacheLoader">
            <properties>
               pushStateWhenCoordinator=true
               pushStateWhenCoordinatorTimeout=20000
            </properties>
         </singletonStore>
      </loader>
   </loaders>

   <!--
      Define custom interceptors.  All custom interceptors need to extend
org.jboss.cache.interceptors.base.CommandInterceptor
   -->
   <!--
   <customInterceptors>
      <interceptor position="first"

class="org.jboss.cache.config.parsing.custominterceptors.AaaCustomInterceptor">
         <property name="attrOne" value="value1" />
         <property name="attrTwo" value="value2" />
      </interceptor>
      <interceptor position="last"

class="org.jboss.cache.config.parsing.custominterceptors.BbbCustomInterceptor"/>
      <interceptor index="3"

class="org.jboss.cache.config.parsing.custominterceptors.AaaCustomInterceptor"/>
      <interceptor before="org.jboss.cache.interceptors.CallInterceptor"

class="org.jboss.cache.config.parsing.custominterceptors.BbbCustomInterceptor"/>
      <interceptor after="org.jboss.cache.interceptors.CallInterceptor"

class="org.jboss.cache.config.parsing.custominterceptors.AaaCustomInterceptor"/>
   </customInterceptors>
   -->
</jbosscache>
```

12.1.1. XML-Validierung

Konfigurations-XML-Dateien werden mittels eines XSD-Schemas validiert. Dieses Schema ist im **`jbosscache-core.jar`** enthalten und ist zudem online erhältlich: **`http://www.jboss.org/jbosscache/jbosscache-config-3.0.xsd`**. Die meisten IDEs und XML-Authoring-Werkzeuge können mit diesem Schema umgehen und so Ihre Konfigurationsdatei schon

beim Schreiben validieren.

JBoss Cache überprüft beim Start ebenfalls Ihre Konfigurationsdatei und meldet ggf. eine Ausnahme, falls es eine ungültige Datei feststellt. Sie können dieses Verhalten unterdrücken, indem Sie beim Start **`-Djbosscache.config.validate=false`** an ihre JVM übergeben. Alternativ können Sie den Validator auf ein anderes Schema verweisen, indem Sie **`-Djbosscache.config.schemaLocation=url`** übergeben.

12.2. Kurzanleitung zur Konfigurationsdatei

Sehen Sie nachfolgend eine Liste mit Definitionen aller oben erwähnten XML-Elementattribute sowie deren Bean-Äquivalente zur befehlsorientierten Konfiguration. Besagt die Beschreibung eines Attributs, dass dieses *dynamisch* ist, bedeutet das, dass dieses Attribut nach dem Erzeugen und Starten des Caches noch geändert werden kann.

Tabelle 12.1. Das `<jbosscache />`-Element

Das `<jbosscache />`-Element	
Beschreibung	Dies ist das Wurzelelement für die JBoss Cache Konfigurationsdatei. Dies ist das einzige obligatorische Element in einer gültigen JBoss Cache Konfigurationsdatei.
Übergeordnetes Element	keines (dies ist das Wurzelelement)
Untergeordnete Elemente	Tabelle 12.40, „Das **`<clustering />`**-Element“, Tabelle 12.37, „Das **`<customInterceptors />`**-Element“, Tabelle 12.19, „Das **`<eviction />`**-Element“, Tabelle 12.15, „Das **`<invocationBatching />`**-Element“, Tabelle 12.7, „Das **`<jmxStatistics />`**-Element“, Tabelle 12.13, „Das **`<listeners />`**-Element“, Tabelle 12.27, „Das **`<loaders />`**-Element“, Tabelle 12.3, „Das **`<locking />`**-Element“, Tabelle 12.17, „Das **`<serialization />`**-Element“, Tabelle 12.11, „Das **`<shutdown />`**-Element“, Tabelle 12.9, „Das **`<startup />`**-Element“, Tabelle 12.5, „Das **`<transaction />`**-Element“
Bean-Äquivalent	**`Configuration`**

Tabelle 12.2. `<jbosscache />`-Attribute

`<jbosscache />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
xmlns	-	urn:jboss:jbosscache-core:config:3.1	urn:jboss:jbosscache-core:config:3.1	Definiert den XML-Namensraum für alle Konfigurationseinträge.
xmlns:xsi	-	http://www.w3.org/2001/XMLSchema-instance	http://www.w3.org/2001/XMLSchema-instance	Definiert die XML-Schemainstanz für die Konfiguration.

Tabelle 12.3. Das `<locking />`-Element

Das `<locking />`-Element	
Beschreibung	Dieses Element spezifiziert das Sperrverhalten des Caches.
Übergeordnetes Element	Tabelle 12.1, „Das **`<jbosscache />`**-Element“
Untergeordnete Elemente	
Bean-Äquivalent	**`Configuration`**

Tabelle 12.4. `<locking />`-Attribute

`<locking />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
isolationLevel	isolationLevel	READ_COMMITTED, REPEATABLE_READ	REPEATABLE_READ	Die für Transaktionen verwendete Isolationsebene.
lockParentForChildInsertRemove	lockParentForChildInsertRemove	true, false	false	Legt fest, ob übergeordnete Knoten gesperrt werden, wenn untergeordnete Knoten eingefügt oder entfernt werden. Dies kann auch je Knoten konfiguriert werden (siehe **`Node.setLockForChildInsertRemove()`**)
lockAcquisitionTimeout	lockAcquisitionTimeout (*dynamisch*)	Jeder positive Long-Wert	10000	Zeit in Millisekunden, die ein Thread versucht, eine Sperre zu erhalten. In der Regel wird eine **`TimeoutException`** ausgegeben, wenn in dieser Zeit keine Sperre erhalten werden kann. Kann für einzelne Aufrufe mittels **`Option.setLockAcquisitionTimeout()`** außer Kraft gesetzt werden.
nodeLockingScheme (*veraltet*)	nodeLockingScheme	mvcc, pessimistisch, optimistisch	mvcc	Spezifiziert das zu verwendende Knoten-Sperrschema.
writeSkewCheck	writeSkewCheck	true, false	false	Legt fest, ob auf Write Skews hin überprüft werden soll. Wird nur

				verwendet, wenn **`nodeLockingScheme`** auf **`mvcc`** und **`isolationLevel`** auf **`REPEATABLE_READ`** gesetzt ist. Siehe Abschnitt 11.1.1.2.2, „Nebenläufige Schreibvorgänge und Write-Skews“ für weitere Einzelheiten.
useLockStriping	useLockStriping	true, false	true	Legt fest, ob Lock Striping angewendet wird. Wird nur verwendet, wenn **`nodeLockingScheme`** auf **`mvcc`** gesetzt ist. Lock Striping bietet in der Regel bessere Leistung und Speicherverbrauch, kann jedoch unter gewissen Umständen gegenseitige Sperren verursachen, wenn mehrere FQNs auf dieselbe, gemeinsam verwendete Sperre verweisen. Durch Erhöhen Ihrer Nebenläufigkeitsebene kann dieses Problem gelindert werden, vollständig abstellen lässt es sich jedoch nur durch Deaktivieren von Lock Striping.
concurrencyLev	concurrencyLevel	Jeder positive,	500	Spezifiziert die

el	ganzzahlige Wert; 0 ist unzulässig.	Anzahl gemeinsamer Sperren, die für erhaltene Schreibsperren zu verwenden sind. Wird nur verwendet, wenn **`nodeLockingScheme`** auf **`mvcc`** gesetzt ist. Siehe Abschnitt 11.1.1.2, „MVCC-Implementierung“ für nähere Erläuterungen.

Tabelle 12.5. Das `<transaction />`-Element

Das `<transaction />`-Element	
Beschreibung	Dieses Element spezifiziert das transaktionale Verhalten des Caches.
Übergeordnetes Element	Tabelle 12.1, „Das **`<jbosscache />`**-Element“
Untergeordnete Elemente	
Bean-Äquivalent	**`Configuration`**

Tabelle 12.6. `<transaction />`-Attribute

`<transaction />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
transactionManagerLookupClass	transactionManagerLookupClass	Eine gültige Klasse, die im Klassenpfad verfügbar ist	keiner	Legt die **`TransactionManagerLookupClass`**-Implementierung fest, mit Hilfe derer ein Transaktionsmanager bezogen werden soll. Falls nicht spezifiziert (und kein **`TransactionManager`** mittels **`RuntimeConfig.setTransactionManager()`** injiziert wird), wird der Cache an keiner Transaktion teilnehmen können.
syncCommitPhase	syncCommitPhase (*dynamisch*)	true, false	false	Falls aktiviert, werden die im Cluster versendeten Festschreibungsnachrichten auf synchrone Weise verschickt. Dies ist in der Regel kaum von Vorteil, denn nach Entdecken eines Fehler bei der Übertragung der Festschreibung kann nichts weiter unternommen werden, als eine entsprechende Meldungs ins Protokoll zu schreiben, da einige Knoten im Cluster womöglich

				bereits festgeschrieben haben und nicht mehr zurücksetzen können.
syncRollbackPhase	syncRollbackPhase (*dynamisch*)	true, false	false	Falls aktiviert, werden die im Cluster versendeten Zurücksetzungsnachrichten ("Rollback") auf synchrone Weise verschickt. Dies ist in der Regel kaum von Vorteil, denn nach Entdecken eines Fehler bei der Übertragung des Rollbacks kann nichts weiter unternommen werden, als eine entsprechende Meldungs ins Protokoll zu schreiben, da einige Knoten im Cluster womöglich bereits festgeschrieben haben und nicht mehr zurücksetzen können.

Tabelle 12.7. Das `<jmxStatistics />`-Element

Das `<jmxStatistics />`-Element	
Beschreibung	Dieses Element spezifiziert, ob Cache-Statistiken gesammelt und via JMX berichtet werden.
Übergeordnetes Element	Tabelle 12.1, „Das **`<jbosscache />`**-Element“
Untergeordnete Elemente	
Bean-Äquivalent	**`Configuration`**

Tabelle 12.8. `<jmxStatistics />`-Attribute

`<jmxStatistics />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
enabled	exposeManagementStatistics	true, false	true	Steuert, ob Cache-Statistiken gesammelt und via JMX berichtet werden.

Tabelle 12.9. Das `<startup />`-Element

Das `<startup />`-Element	
Beschreibung	Dieses Element spezifiziert das Verhalten beim Start des Caches.
Übergeordnetes Element	Tabelle 12.1, „Das **`<jbosscache />`**-Element“
Untergeordnete Elemente	
Bean-Äquivalent	**`Configuration`**

Tabelle 12.10. `<startup />`-Attribute

`<startup />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
regionsInactiveOnStartup	inactiveOnStartup	true, false	false	Sind Bereiche (siehe Abschnitt 7.6, „Klassenladen und Bereiche“) aktiviert, steuert dieses Attribut, ob neu erzeugte Bereiche beim Start aktiv sind.

Tabelle 12.11. Das `<shutdown />`-Element

Das `<shutdown />`-Element	
Beschreibung	Dieses Element spezifiziert das Verhalten beim Beenden des Caches.
Übergeordnetes Element	Tabelle 12.1, „Das **`<jbosscache />`**-Element“
Untergeordnete Elemente	
Bean-Äquivalent	**`Configuration`**

Tabelle 12.12. <shutdown />-Attribute

<shutdown />-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
hookBehavior	shutdownHookBehavior	DEFAULT, DONT_REGISTER, REGISTER	DEFAULT	Dieses Attribut bestimmt, ob der Cache einen JVM Shutdown Hook registrieren soll, damit Ressourcen bereinigt werden können, falls die JVM ein Shutdown-Signal empfängt. Standardmäßig ist ein Shutdown Hook registriert, falls kein MBean-Server (außer dem JDK-Standard) entdeckt wird. REGISTER zwingt den Cache zur Registrierung eines Shutdown Hooks, selbst wenn ein MBean-Server gefunden wird, und DONT_REGISTER zwingt den Cache, KEINEN Shutdown Hook zu registrieren, selbst wenn kein MBean-Server gefunden wird.

Tabelle 12.13. Das <listeners />-Element

Das <listeners />-Element	
Beschreibung	Dieses Element spezifiziert das Verhalten von registrierten Cache-Listenern.
Übergeordnetes Element	Tabelle 12.1, „Das **<jbosscache />**-Element“
Untergeordnete Elemente	
Bean-Äquivalent	**Configuration**

Tabelle 12.14. `<listeners />`-Attribute

`<listeners />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
asyncPoolSize	listenerAsyncPoolSize	Ganzzahl	1	Die Größe des Thread-Pools, der zur Verteilung von Ereignissen auf Cache-Listener genutzt wird, die sich als asynchrone Listener registriert haben. Ist diese Zahl kleiner als 1, werden alle asynchronen Listener als synchrone Listener behandelt und synchron benachrichtigt.
asyncQueueSize	listenerAsyncQueueSize	positive Ganzzahl	50000	Die Größe der abgegrenzten Queue, die vom asynchronen Listener Thread-Pool genutzt wird. Wird nur berücksichtigt, wenn **`asyncPoolSize`** größer als 0 ist. Vergrößern Sie diese, wenn Sie feststellen, dass viele Threads sperren beim Versuch, Ereignisse zu dieser Queue hinzuzufügen.

Tabelle 12.15. Das `<invocationBatching />`-Element

Das `<invocationBatching />`-Element	
Beschreibung	Dieses Element spezifiziert das Verhalten des Aufruf-Batchings.
Übergeordnetes Element	Tabelle 12.1, „Das `<jbosscache />`-Element“
Untergeordnete Elemente	
Bean-Äquivalent	**`Configuration`**

Tabelle 12.16. `<invocationBatching />`-Attribute

`<invocationBatching />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
enabled	invocationBatchingEnabled	true, false	false	Ob Aufruf-Batching aktiviert ist oder nicht. Siehe Kapitel 4, *Batching-API* für Einzelheiten.

Tabelle 12.17. Das `<serialization />`-Element

Das `<serialization />`-Element	
Beschreibung	Dieses Element spezifiziert das Verhalten der Objektserialisierung in JBoss Cache.
Übergeordnetes Element	Tabelle 12.1, „Das `<jbosscache />`-Element“
Untergeordnete Elemente	
Bean-Äquivalent	**`Configuration`**

Tabelle 12.18. `<serialization />`-Attribute

`<serialization />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
marshallerClass	marshallerClass	Eine gültige Klasse, die im Klassenpfad verfügbar ist	VersionAwareMarshaller	Spezifiziert den zu verwendenden Marshaller beim Serialisieren und Deserialisieren von Objekten, ob zur Replikation oder Persistierung.
useLazyDeserialization	useLazyDeserialization	true, false	false	Ein Verfahren, bei dem Serialisierung und Deserialisierung von Objekten verzögert wird, bis diese tatsächlich verwendet und benötigt werden. Das bedeutet in der Regel, dass jede Deserialisierung unter Verwendung des Thread-Kontext-Klassenladers desjenigen Aufrufs erfolgt, der die Deserialisierung erfordert, was ein effektives Verfahren zur Klassenlader-Isolation ist.
useRegionBasedMarshalling (*veraltet*)	useRegionBasedMarshalling	true, false	false	Ein älteres Verfahren, durch das Klassenlader-Isolation erreicht wurde, indem Klassenlader für bestimmte Bereiche registriert wurden.
version	replicationVersion	Gültiger JBoss Cache Versions-String	Aktuelle Version	Verwendet vom **`VersionAwareMarshaller`** beim

				Bestimmen, welcher Version-Stream-Parser standardmäßig zu verwenden ist bei der Iniziierung der Kommunikation in einem Cluster. Nützlich, wenn Sie eine neuere Version von JBoss Cache in einem Cluster einsetzen müssen, der noch ältere Versionen enthält, und kann zum Durchführen von Rolling Upgrades verwendet werden.
objectInputStreamPoolSize	objectInputStreamPoolSize	Positive Ganzzahl	50	Derzeit nicht verwendet.
objectOutputStreamPoolSize	objectOutputStreamPoolSize	Positive Ganzzahl	50	Derzeit nicht verwendet.

Tabelle 12.19. Das `<eviction />`-Element

Das `<eviction />`-Element	
Beschreibung	Dieses Element steuert, wie Eviction in dem Cache funktioniert.
Übergeordnetes Element	Tabelle 12.1, „Das **`<jbosscache />`**-Element“
Untergeordnete Elemente	Tabelle 12.21, „Das **`<default />`**-Element“, Tabelle 12.23, „Das **`<region />`**-Element“
Bean-Äquivalent	**`EvictionConfig`**

Tabelle 12.20. `<eviction />`-Attribute

`<eviction />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
wakeUpInterval	wakeupInterval	Ganzzahl	5000	Die Häufigkeit, mit der der Eviction-Thread ausgeführt wird, in Millisekunden. Falls auf weniger als 1 eingestellt, wird der Eviction-Thread nie ausgeführt und ist damit faktisch deaktiviert.

Tabelle 12.21. Das `<default />`-Element

Das `<default />`-Element	
Beschreibung	Dieses Element definiert den standardmäßigen Eviction-Bereich.
Übergeordnetes Element	Tabelle 12.19, „Das **`<eviction />`**-Element“
Untergeordnete Elemente	Tabelle 12.25, „Das **`<property />`**-Element“
Bean-Äquivalent	**`EvictionRegionConfig`**

Tabelle 12.22. `<default />`-Attribute

`<default />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
algorithmClass	evictionAlgorithmConfig	Eine gültige Klasse, die im Klassenpfad verfügbar ist	keiner	Dieses Attribut muss spezifiziert werden, falls dieser Tag verwendet wird. Beachten Sie, dass Sie bei der befehlsorientierten Konfiguration die dem Eviction-Algorithmus entsprechende **`EvictionAlgorithmConfig`**-Datei benutzen sollten. Wenn Sie beispielsweise **`LRUAlgorithm`** in XML verwenden, würden Sie befehlsorientiert eine Instanz von **`LRUAlgorithmConfig`** nutzen.
actionPolicyClass	evictionActionPolicyClassName	Eine gültige Klasse, die im Klassenpfad verfügbar ist	DefaultEvictionActionPolicy	Die Richtlinienklasse der Eviction-Aktion; definiert, was geschehen soll, wenn ein Knoten geräumt werden muss.
eventQueueSize	eventQueueSize (*dynamisch*)	Ganzzahl	200000	Die Größe der abgegrenzten Eviction-Ereignis-Queue.

Tabelle 12.23. Das `<region />`-Element

Das `<region />`-Element	
Beschreibung	Dieses Element definiert einen Eviction-Bereich. Mehrere Instanzen dieses Tags können existieren, vorausgesetzt, sie besitzen eindeutige **`name`**-Attribute.
Übergeordnetes Element	Tabelle 12.19, „Das **`<eviction />`**-Element“
Untergeordnete Elemente	Tabelle 12.25, „Das **`<property />`**-Element“
Bean-Äquivalent	**`EvictionRegionConfig`**

Tabelle 12.24. `<region />`-Attribute

`<region />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
name	regionFqn	Ein String, der mittels Fqn.fromString() analysiert werden könnte	keiner	Dies sollte ein eindeutiger Name sein, der diesen Bereich definiert. Siehe Abschnitt 10.2, „Eviction-Bereiche“ für Einzelheiten über Eviction-Bereiche.
algorithmClass	evictionAlgorithmConfig	Eine gültige Klasse, die im Klassenpfad verfügbar ist	keiner	Dieses Attribut muss spezifiziert werden, falls dieser Tag verwendet wird. Beachten Sie, dass Sie bei der befehlsorientierten Konfiguration die dem Eviction-Algorithmus entsprechende **`EvictionAlgorithmConfig`**-Datei benutzen sollten. Wenn Sie beispielsweise **`LRUAlgorithm`** in XML verwenden, würden Sie befehlsorientiert eine Instanz von **`LRUAlgorithmConfig`** nutzen.
actionPolicyClass	evictionActionPolicyClassName	Eine gültige Klasse, die im Klassenpfad verfügbar ist	DefaultEvictionActionPolicy	Die Richtlinienklasse der Eviction-Aktion; definiert, was geschehen soll, wenn ein Knoten geräumt werden muss.
eventQueueSize	eventQueueSize (*dynamisch*)	Ganzzahl	200000	Die Größe der abgegrenzten Eviction-Ereignis-Queue.

Tabelle 12.25. Das `<property />`-Element

Das `<property />`-Element	
Beschreibung	Ein Verfahren zum Übergeben von Name-Wert-Eigenschaften an das umgebene Konfigurationselement.
Übergeordnetes Element	Tabelle 12.21, „Das **`<default />`**-Element“, Tabelle 12.23, „Das **`<region />`**-Element“, Tabelle 12.38, „Das **`<interceptor />`**-Element“
Untergeordnete Elemente	
Bean-Äquivalent	Entweder direkte Setter oder **`setProperties()`**-umfassende Bean

Tabelle 12.26. `<property />`-Attribute

`<property />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
name	Entweder direkte Setter oder **`setProperties()`**-umfassende Bean	String	keiner	Eigenschaftsname
value	Entweder direkte Setter oder **`setProperties()`**-umfassende Bean	String	keiner	Eigenschaftswert

Tabelle 12.27. Das `<loaders />`-Element

Das `<loaders />`-Element	
Beschreibung	Definiert jegliche Cache Loader.
Übergeordnetes Element	Tabelle 12.1, „Das **`<jbosscache />`**-Element“
Untergeordnete Elemente	Tabelle 12.29, „Das **`<preload />`**-Element“, Tabelle 12.32, „Das **`<loader />`**-Element“
Bean-Äquivalent	CacheLoaderConfig

Tabelle 12.28. `<loaders />`-Attribute

`<loaders />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
passivation	passivation	true, false	false	Falls "true", werden Cache Loader im Passivierungsmodus verwendet. Siehe Kapitel 9, *Cache Loader* für eine ausführliche Erläuterung diesbezüglich.
shared	shared	true, false	false	Falls "true", werden Cache Loader gemeinsam verwendet. Siehe Kapitel 9, *Cache Loader* für eine ausführliche Erläuterung diesbezüglich.

Tabelle 12.29. Das `<preload />`-Element

Das `<preload />`-Element	
Beschreibung	Definiert das Vorladen von FQN-Unterbäumen beim Start des Caches. Dieses Element besitzt keinerlei Attribute.
Übergeordnetes Element	Tabelle 12.27, „Das **`<loaders />`**-Element“
Untergeordnete Elemente	Tabelle 12.30, „Das **`<node />`**-Element“
Bean-Äquivalent	CacheLoaderConfig

Tabelle 12.30. Das `<node />`-Element

Das `<node />`-Element	
Beschreibung	Dieses Element definiert einen Unterbaum, unter dem alle Inhalte von den Cache Loadern beim Start des Caches vorgeladen werden. Es können mehrere Unterbäume vorgeladen werden, allerdings macht es nur dann Sinn mehrere Unterbäume zu definieren, wenn diese sich nicht überschneiden.
Übergeordnetes Element	Tabelle 12.29, „Das **`<preload />`**-Element“
Untergeordnete Elemente	
Bean-Äquivalent	CacheLoaderConfig

Tabelle 12.31. <node />-Attribute

<node />-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
fqn	preload	String	keiner	Ein vorzuladender FQN. Dies sollte ein String sein, der mittels Fqn.fromString() analysiert werden kann. Erfolgt dies befehlsorientiert, sollten Sie einen einzigen String erzeugen, der alle FQNs, die Sie vorladen möchten, durch Leerzeichen voneinander getrennt enthält, und übergeben diesen an **`CacheLoaderConfig.setPreload()`**.

Tabelle 12.32. Das <loader />-Element

Das <loader />-Element	
Beschreibung	Dieses Element definiert einen Cache Loader. Es können mehrere Elemente verwendet werden, um Cache-Loader-Ketten zu erzeugen.
Übergeordnetes Element	Tabelle 12.27, „Das **<loaders />**-Element“
Untergeordnete Elemente	Tabelle 12.34, „Das **<properties />**-Element“, Tabelle 12.35, „Das **<singletonStore />**-Element“
Bean-Äquivalent	IndividualCacheLoaderConfig

Tabelle 12.33. `<loader />`-Attribute

`<loader />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
class	className	Eine gültige Klasse, die im Klassenpfad verfügbar ist	keiner	Eine zu verwendende Cache-Loader-Implementierung.
async	async	true, false	false	Alle Veränderungen an diesem Cache Loader erfolgen asynchron auf einem separaten Thread.
fetchPersistentState	fetchPersistentState	true, false	false	Wenn ein Cache startet, rufe den persistenten Zustand von den Cache Loadern anderer Caches im Cluster ab. Nur *ein* Loader-Element darf dies auf "true" setzen. Zudem ist dies nur dann sinnvoll, wenn der Tabelle 12.40, „Das `<clustering />`-Element“-Tag vorhanden ist.
purgeOnStartup	purgeOnStartup	true, false	false	Bereinigt diesen Cache Loader beim Start.

Tabelle 12.34. Das `<properties />`-Element

Das `<properties />`-Element	
Beschreibung	Dieses Element enthält eine Reihe von Eigenschaften, die von einer **`java.util.Properties`**-Instanz gelesen werden können. Dieser Tag hat keine Attribute, und die Inhalte dieses Tags werden von **`Properties.load()`** analysiert.
Übergeordnetes Element	Tabelle 12.32, „Das **`<loader />`**-Element“, Tabelle 12.35, „Das **`<singletonStore />`**-Element“, Tabelle 12.52, „Das **`<locator />`**-Element“
Untergeordnete Elemente	
Bean-Äquivalent	IndividualCacheLoaderConfig.setProperties()

Tabelle 12.35. Das `<singletonStore />`-Element

Das `<singletonStore />`-Element	
Beschreibung	Dieses Element konfiguriert den umgebenden Cache Loader als eine Abschnitt 9.2.1, „Singleton-Store-Konfiguration“.
Übergeordnetes Element	Tabelle 12.32, „Das **`<loader />`**-Element“
Untergeordnete Elemente	Tabelle 12.34, „Das **`<properties />`**-Element“
Bean-Äquivalent	SingletonStoreConfig

Tabelle 12.36. `<singletonStore />`-Attribute

`<singletonStore />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
class	className	Eine gültige Klasse, die im Klassenpfad verfügbar ist	SingletonStoreCacheLoader	Eine zu verwendende Singleton Store Wrapper-Implementierung.
enabled	enabled	true, false	false	Falls "true", ist der Singleton Store Cache Loader aktiviert.

Tabelle 12.37. Das `<customInterceptors />`-Element

Das `<customInterceptors />`-Element	
Beschreibung	Dieses Element ermöglicht Ihnen das Definieren angepasster Interzeptoren für den Cache. Dieser Tag hat keine Attribute.
Übergeordnetes Element	Tabelle 12.1, „Das **`<jbosscache />`**-Element“
Untergeordnete Elemente	Tabelle 12.38, „Das **`<interceptor />`**-Element“
Bean-Äquivalent	Keines. Instanziieren Sie zur Laufzeit Ihren eigenen Interzeptor und übergeben diesen mittels **`Cache.addInterceptor()`** an den Cache.

Tabelle 12.38. Das `<interceptor />`-Element

Das `<interceptor />`-Element	
Beschreibung	Dieses Element ermöglicht Ihnen das Konfigurieren eines angepassten Interzeptors. Dieser Tag darf mehrmals erscheinen.
Übergeordnetes Element	Tabelle 12.37, „Das **`<customInterceptors />`**-Element“
Untergeordnete Elemente	Tabelle 12.25, „Das **`<property />`**-Element“
Bean-Äquivalent	Keines. Instanziieren Sie zur Laufzeit Ihren eigenen Interzeptor und übergeben diesen mittels **`Cache.addInterceptor()`** an den Cache.

Tabelle 12.39. `<interceptor />`-Attribute

`<interceptor />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
class	-	Eine gültige Klasse, die im Klassenpfad verfügbar ist	keiner	Eine Implementierung von **`CommandInterceptor`**.
position	-	first, last		Die Position, an der dieser Interzeptor in der Kette platziert wird. "first" ist der erste Interzeptor, der gefunden wird, wenn ein Aufruf auf dem Cache erfolgt, "last" ist der letzte Interzeptor, bevor der Aufruf an die Datenstruktur übergeben wird. *Beachten Sie, dass dieses Attribut und **`before`**, **`after`** und **`index`** sich gegenseitig ausschließen.*
before	-	Vollqualifizierter Klassenname eines Interzeptors		Platziert den neuen Interzeptor direkt vor der Instanz des benannten Interzeptors. *Beachten Sie, dass dieses Attribut und **`position`**, **`after`** und **`index`** sich gegenseitig ausschließen.*
after	-	Vollqualifizierter Klassenname eines Interzeptors		Platziert den neuen Interzeptor direkt nach der Instanz des benannten Interzeptors. *Beachten Sie,*

			*dass dieses Attribut und **`position`**, **`before`** und **`index`** sich gegenseitig ausschließen.*
index	-	Positive Ganzzahl	Die Position, an der dieser Interzeptor in der Kette platziert wird, wobei 0 die erste Position ist. *Beachten Sie, dass dieses Attribut und **`position`**, **`before`** und **`after`** gegenseitig ausschließen.*

Tabelle 12.40. Das `<clustering />`-Element

Das `<clustering />`-Element	
Beschreibung	Ist dieses Element vorhanden, wird der Cache im geclusterten Modus gestartet. Attribute und Unterelement definieren die Clustering-Charakteristiken.
Übergeordnetes Element	Tabelle 12.1, „Das **`<jbosscache />`**-Element“
Untergeordnete Elemente	Tabelle 12.46, „Das **`<stateRetrieval />`**-Element“, Tabelle 12.42, „Das **`<sync />`**-Element“, Tabelle 12.44, „Das **`<async />`**-Element“, Tabelle 12.48, „Das **`<buddy />`**-Element“, Tabelle 12.54, „Das **`<jgroupsConfig />`**-Element“
Bean-Äquivalent	Konfiguration

Tabelle 12.41. `<clustering />`-Attribute

`<clustering />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
mode	cacheMode	replication, invalidation, r, i	replication	Siehe Kapitel 8, *Cache-Modi und Clustering* für die Unterschiede zwischen Replikation und Invalidierung. Wird die Bean verwendet, wird synchrone und asynchrone Kommunikation mit dem Clustering-Modus kombiniert, um Ihnen die **`Configuration.CacheMode`**-Enumeration zu liefern.
clusterName	clusterName	String	JBossCache-cluster	Ein Cluster-Name, um den beizutretenden Cluster zu identifizieren.

Tabelle 12.42. Das `<sync />`-Element

Das `<sync />`-Element	
Beschreibung	Ist dieses Element vorhanden, erfolgt sämtliche Kommunikation synchron, d.h., sobald ein Thread eine Nachricht über das Netzwerk sendet, sperrt dieser, bis er eine Bestätigung vom Empfänger erhält. Dieses Element und das Tabelle 12.44, „Das **`<async />`**-Element“ schließen sich gegenseitig aus.
Übergeordnetes Element	Tabelle 12.40, „Das **`<clustering />`**-Element“
Untergeordnete Elemente	
Bean-Äquivalent	Configuration.setCacheMode()

Tabelle 12.43. <sync />-Attribute

<sync />-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
replTimeout	syncReplTimeout (*dynamisch*)	positive Ganzzahl	15000	Dies ist die Zeit, die bei einem entfernten Aufruf auf eine Bestätigung gewartet wird, bevor eine Ausnahme geworfen wird.

Tabelle 12.44. Das <async />-Element

Das <async />-Element	
Beschreibung	Ist dieses Element vorhanden, erfolgt sämtliche Kommunikation asynchron, d.h., wenn ein Thread eine Nachricht über das Netzwerk sendet, wartet dieser nicht auf eine Bestätigung, bevor er zurückkehrt. Dieses Element und das Tabelle 12.42, „Das **<sync />**-Element“ schließen sich gegenseitig aus.
Übergeordnetes Element	Tabelle 12.40, „Das **<clustering />**-Element“
Untergeordnete Elemente	
Bean-Äquivalent	Configuration.setCacheMode()

Tabelle 12.45. <async />-Attribute

<async />-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
serializationExecutorPoolSize	serializationExecutorPoolSize	positive Ganzzahl	25	Zusätzlich zur asynchron durchgeführten Replikation erfolgt sogar die Serialisierung der Inhalte für die Replikation in einem separaten Thread, damit der Aufrufer so schnell wie möglich zurückkehren kann. Diese Einstellung steuert die Größe des Thread-Pools zur Serialisierung. Die Einstellung auf einen Wert kleiner als 1 bedeutet, dass die Serialisierung nicht asynchron erfolgt.
serializationExecutorQueueSize	serializationExecutorQueueSize	positive Ganzzahl	50000	Hiermit wird die Größe der abgegrenzten Queue definiert, die Aufgaben für den Serialization Executor enthält. Dies wird ignoriert, falls kein Serialization Executor verwendet wird, wenn also z.B. **serializationExecutorPoolSize** kleiner als 1 ist.
useReplQueue	useReplQueue	true, false	false	Falls "true", zwingt dies alle asynchrone Kommunikation in

				eine Warteschlange, die dann regelmäßig gebündelt versendet wird.
replQueueInterval	replQueueInterval	positive Ganzzahl	5000	Ist **useReplQueue** auf "true" gesetzt, steuert dieses Attribut, wie oft der anynchrone Thread, der zum Leeren der Replikations-Queue genutzt wird, ausgeführt wird. Dies sollte eine positive Ganzzahl sein, welche die Thread-Wakeup-Zeit in Millisekunden darstellt.
replQueueMaxElements	replQueueMaxElements	positive Ganzzahl	1000	Ist **useReplQueue** auf "true" gesetzt, kann dieses Attribut dazu verwendet werden, das Leeren der Queue auszulösen, sobald diese eine bestimmte Schwelle überschreitet.

Tabelle 12.46. Das `<stateRetrieval />`-Element

Das `<stateRetrieval />`-Element	
Beschreibung	Dieser Tag steuert, wie Zustände von benachbarten Caches abgerufen werden, wenn diese Cache-Instanz startet.
Übergeordnetes Element	Tabelle 12.40, „Das **`<clustering />`**-Element“
Untergeordnete Elemente	
Bean-Äquivalent	Konfiguration

Tabelle 12.47. `<stateRetrieval />`-Attribute

`<stateRetrieval />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
fetchInMemoryState	fetchInMemoryState	true, false	true	Falls "true", fragt der Cache beim Start den Zustand von benachbarten Caches ab, so dass der Cache einen "Warmstart" hat.
timeout	stateRetrievalTimeout	positive Ganzzahl	10000	Dies ist die maximale Zeit in Millisekunden, die auf den Zustand von benachbarten Caches gewartet wird, bevor eine Ausnahme ausgegeben und der Start abgebrochen wird.
nonBlocking	useNonBlockingStateTransfer	true, false	false	Dieser Konfigurationsschalter aktiviert den Mechanismus zur nicht-sperrenden Zustandsübertragung, neu in 3.1.0. Beachten Sie, dass hierzu MVCC als Knotensperrschema erforderlich ist, und dass STREAMING_STATE_TRANSFER im verwendeten JGroups-Stapel vorhanden sein muss.

Tabelle 12.48. Das <buddy />-Element

Das <buddy />-Element	
Beschreibung	Ist dieser Tag vorhanden, wird der Zustand nicht über den gesamten Cluster repliziert. Stattdessen wird Buddy-Replikation eingesetzt, um auf ausgewählten Cache-Instanzen Backups zu bewahren. Siehe Abschnitt 8.1.2.2, „Buddy-Replikation“ für Details. Beachten Sie, dass dies nur dann verwendet wird, wenn der Clustering-Modus **`replication`** ist, nicht aber bei **`invalidation`**.
Übergeordnetes Element	Tabelle 12.40, „Das **`<clustering />`**-Element“
Untergeordnete Elemente	Tabelle 12.50, „Das **`<dataGravitation />`**-Element“, Tabelle 12.52, „Das **`<locator />`**-Element“,
Bean-Äquivalent	BuddyReplicationConfig

Tabelle 12.49. <buddy />-Attribute

<buddy />-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
enabled	enabled	true, false	false	Falls "true", ist die Buddy-Replikation aktiviert.
communicationTimeout	buddyCommunicationTimeout	positive Ganzzahl	10000	Dies ist die maximale Zeit in Millisekunden, die auf Kommunikation zur Buddy-Gruppenorganisation von Buddy-Caches gewartet wird.
poolName	buddyPoolName	String		Dies wird als Mittel eingesetzt, um Cache-Instanzen zu identifizieren und um Hinweise an die Algorithmen zur Buddy-Replikation zu geben. Mehr dazu im Abschnitt 8.1.2.2, „Buddy-Replikation“.

Tabelle 12.50. Das `<dataGravitation />`-Element

Das `<dataGravitation />`-Element	
Beschreibung	Dieser Tag konfiguriert, wie Datengravitation abgewickelt wird. Siehe Abschnitt 8.1.2.2, „Buddy-Replikation“ für Details.
Übergeordnetes Element	Tabelle 12.48, „Das **`<buddy />`**-Element“
Untergeordnete Elemente	
Bean-Äquivalent	BuddyReplicationConfig

Tabelle 12.51. `<dataGravitation />`-Attribute

`<dataGravitation />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
auto	autoDataGravitation	true, false	true	Falls "true", wird, wenn eine get() auf einem Cache ausgeführt wird und dabei nichts gefunden wird, die Datengravitation von benachbarten Caches versucht. Ist dies "false", kann eine Gravitation nur erfolgen, wenn die **`Option.setForceDataGravitation()`**-Option angegeben ist.
removeOnFind	dataGravitationRemoveOnFind	true, false	true	Falls "true", wird bei der Gravitation die Instanz, die die Gravitation anfragt, nun zum Besitzer des abgefragten Zustands und fordert alle anderen Instanzen dazu auf, diesen Zustand aus ihrem Speicher zu entfernen.
searchBackupTrees	dataGravitationSearchBackupTrees	true, false	true	Falls "true", werden eingehende Gravitationsanfragen den Cache dazu veranlassen, nicht nur seine primäre Datenstruktur zu durchsuchen, sondern auch seine Backup-Struktur.

Tabelle 12.52. Das `<locator />`-Element

Das `<locator />`-Element	
Beschreibung	Dieser Tag bietet einen zuschaltbaren Mechanismus, um Buddy-Location-Algorithmen bereitzustellen.
Übergeordnetes Element	Tabelle 12.48, „Das **`<buddy />`**-Element“
Untergeordnete Elemente	Tabelle 12.34, „Das **`<properties />`**-Element“
Bean-Äquivalent	BuddyLocatorConfig

Tabelle 12.53. `<locator />`-Attribute

`<locator />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
class	className	Eine gültige Klasse, die im Klassenpfad verfügbar ist	NextMemberBuddyLocator	Eine **`BuddyLocator`**-Implementierung, die zur Auswahl von Buddys aus dem Cluster verwendet werden soll. Siehe **`BuddyLocator`**-Javadocs für Details.

Tabelle 12.54. Das `<jgroupsConfig />`-Element

Das `<jgroupsConfig />`-Element	
Beschreibung	Dieser Tag bietet eine Konfiguration, die mit JGroups verwendet wird, um einen Netzwerkkommunikations-Channel zu erzeugen.
Übergeordnetes Element	Tabelle 12.40, „Das **`<clustering />`**-Element“
Untergeordnete Elemente	Eine Reihe von Elementen, die JGroups-Protokolle repräsentieren (siehe JGroups-Dokumentation). Beachten Sie, dass es keine untergeordneten Elemente gibt, wenn stattdessen Elementattribute verwendet werden. Siehe Abschnitt über Attribute.
Bean-Äquivalent	Konfiguration

Tabelle 12.55. `<jgroupsConfig />`-Attribute

`<jgroupsConfig />`-Attribute				
Attribut	**Bean-Feld**	**Zulässig**	**Standard**	**Beschreibung**
configFile	clusterConfig	Eine JGroups-Konfigurationsdatei im Klassenpfad	udp.xml	Wird dieses Attribut verwendet, werden alle JGroups-Elemente, die Protokolle repräsentieren, innerhalb dieses Tags ignoriert. Stattdessen werden JGroups-Einstellungen von der angegebenen Datei gelesen. Beachten Sie, dass dies nicht zusammen mit dem **`multiplexerStack`**-Attribut verwendet werden kann.
multiplexerStack	muxStackName	Ein gültiger multiplexer Stapelname, der in der Channel Factory existiert, übergeben an die **`RuntimeConfig`**		Dies kann nur mit der **`RuntimeConfig`** verwendet werden, in der Sie mit Hilfe von **`RuntimeConfig.setMuxChannelFactory()`** eine JGroups **`ChannelFactory`**-Instanz übergeben. Wird dieses Attribut verwendet, werden alle JGroups-Elemente, die Protokolle repräsentieren, innerhalb dieses Tags ignoriert. Stattdessen wird der JGroups-Channel mit Hilfe

		der übergebenen Factory erzeugt. Beachten Sie, dass dies nicht zusammen mit dem **`configFile`**-Attribut verwendet werden kann.

Kapitel 13. JMX-Referenzen

13.1. JBoss Cache Statistiken

Eine Fülle von Informationen werden gesammelt und via JMX zur Überwachung des Caches bereitgestellt. Einige davon sind nachfolgend beschrieben:

Tabelle 13.1. JBoss Cache JMX-MBeans

MBean	Attribut-/Operationsname	Beschreibung
DataContainerImpl	getNumberOfAttributes()	Gibt die Anzahl aller Attribute in allen Knoten im Datencontainer aus
	getNumberOfNodes()	Gibt die Anzahl an Knoten im Datencontainer aus
	printDetails()	Zeigt Details des Datencontainers an
RPCManagerImpl	localAddressString	Stringdarstellung der lokalen Adresse
	membersString	Stringdarstellung der Cluster-Ansicht
	statisticsEnabled	Ob RPC-Statistiken gesammelt werden
	replicationCount	Anzahl erfolgreicher Replikationen
	replicationFailures	Anzahl fehlgeschlagener Replikationen
	successRatio	Erfolgsrate von RPC-Aufrufen
RegionManagerImpl	dumpRegions()	Auszug einer Stringdarstellung aller registrierten Bereiche, einschließlich Eviction-Bereiche samt Größe zugehöriger Ereignis-Queues
	numRegions	Anzahl registrierter Bereiche
BuddyManager	buddyGroup	Eine Stringdarstellung der Buddy-Gruppe des Caches
	buddyGroupsIParticipateIn	Stringdarstellungen aller Buddy-Gruppen, an denen der Cache teilnimmt
TransactionTable	numberOfRegisteredTransactions	Anzahl registrierter, laufender Transaktionen
	transactionMap	Eine Stringdarstellung aller derzeit registrierten Transaktionen, die mit internen GlobalTransaction-Instanzen verknüpft sind
MVCCLockManager	concurrencyLevel	Die konfigurierte Nebenläufigkeitsebene
	numberOfLocksAvailable	Anzahl nicht verwendeter Sperren im gemeinsam verwendeten Sperr-Pool
	numberOfLocksHeld	Anzahl verwendeter Sperren im gemeinsam verwendeten Sperr-Pool
	testHashing(String fqn)	Testet die Verteilung von Sperren in den Fqns. Für einen

		angegebenen (stringbasierten) Fqn gibt diese Methode den Index im Sperr-Array zurück, auf das verwiesen wird.
ActivationInterceptor	Aktivierungen	Anzahl passivierter Knoten, die aktiviert wurden.
CacheLoaderInterceptor	CacheLoaderLoads	Anzahl der durch einen Cache Loader geladenen Knoten.
	CacheLoaderMisses	Anzahl erfolgloser Versuche, einen Knoten durch einen Cache Loader zu laden.
CacheMgmtInterceptor	Hits	Anzahl erfolgreicher Attributabrufe.
	Misses	Anzahl erfolgloser Attributabrufe.
	Stores	Anzahl von Attributspeichervorgängen.
	Evictions	Anzahl von Knoten-Räumungen.
	NumberOfAttributes	Anzahl aktuell gecachter Attribute.
	NumberOfNodes	Anzahl aktuell gecachter Knoten.
	ElapsedTime	Anzahl von Sekunden, die der Cache bereits ausgeführt wird.
	TimeSinceReset	Anzahl von Sekunden, seit die Cache-Statistiken zurückgesetzt wurden.
	AverageReadTime	Durchschnittliche Zeit in Millisekunden, um ein Cache-Attribut abzurufen, einschließlich fehlgeschlagener Attributabrufe.
	AverageWriteTime	Durchschnittliche Zeit in Millisekunden zum Schreiben eines Cache-Attributs.
	HitMissRatio	Verhältnis von Treffern zu Treffern und Fehlschlägen. Ein Treffer ist eine "get"-Attributoperation, die zur Folge hat, dass ein Objekt an den Client wiedergegeben wird. Der Abruf kann von einem Cache Loader erfolgen, wenn der Eintrag sich nicht im lokalen Cache befindet.
	ReadWriteRatio	Verhältlis von Lese- zu Schreibvorgängen. Dies ist das Verhältnis von Cache-Treffern und -Fehlschlägen zu Cache-Speicherungen.
CacheStoreInterceptor	CacheLoaderStores	Anzahl in den Cache Loader

		geschriebener Knoten.
InvalidationInterceptor	Invalidierungen	Anzahl von gecachten Knoten, die für ungültig erklärt wurden.
PassivationInterceptor	Passivierungen	Anzahl gecachter Knoten, die passiviert wurden.
TxInterceptor	Prepares	Anzahl von durch diesen Interzeptor durchgeführten Transaktions-"prepare"-Operationen.
	Commits	Anzahl von durch diesen Interzeptor durchgeführten Transaktions-"commit"-Operationen.
	Rollbacks	Anzahl von durch diesen Interzeptor durchgeführten Transaktions-"rollback"-Operationen.
	numberOfSyncsRegistered	Anzahl der beim Transaktionsmanager registrierten Synchronisationen, deren Abschluss und Entfernung noch aussteht.

13.2. JBoss MBean Benachrichtigungen

Die folgende Tabelle zeigt die verfügbaren JMX-Benachrichtigungen für JBoss Cache, sowie die zugehörigen Cache-Ereignisse. Diese Benachrichtigungen können über das **`CacheJmxWrapper`**-MBean empfangen werden. Jede Benachrichtigung repräsentiert ein einzelnes von JBoss Cache veröffentlichtes Ereignis und liefert Benutzerdaten, die mit den Parametern des Ereignisses korrespondieren.

Tabelle 13.2. JBoss Cache MBean Benachrichtigungen

Benachrichtigungstyp	Benachrichtigungsdaten	CacheListener Ereignis
org.jboss.cache.CacheStarted	String: cache service name	@CacheStarted
org.jboss.cache.CacheStopped	String: cache service name	@CacheStopped
org.jboss.cache.NodeCreated	String: fqn, boolean: isPre, boolean: isOriginLocal	@NodeCreated
org.jboss.cache.NodeEvicted	String: fqn, boolean: isPre, boolean: isOriginLocal	@NodeEvicted
org.jboss.cache.NodeLoaded	String: fqn, boolean: isPre	@NodeLoaded
org.jboss.cache.NodeModifed	String: fqn, boolean: isPre, boolean: isOriginLocal	@NodeModifed
org.jboss.cache.NodeRemoved	String: fqn, boolean: isPre, boolean: isOriginLocal	@NodeRemoved
org.jboss.cache.NodeVisited	String: fqn, boolean: isPre	@NodeVisited
org.jboss.cache.ViewChanged	String: view	@ViewChanged
org.jboss.cache.NodeActivated	String: fqn	@NodeActivated
org.jboss.cache.NodeMoved	String: fromFqn, String: toFqn, boolean: isPre	@NodeMoved
org.jboss.cache.NodePassivated	String: fqn	@NodePassivated